NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

AN OPEN ARCHITECTURE FOR DEFENSE VIRTUAL ENVIRONMENT TRAINING SYSTEMS

by

Stephen W. Matthews and Kenneth H. Miller

September 2003

Thesis Advisor:	Rudolph P. Darken
Co-Advisor:	Joseph A. Sullivan

This thesis done in cooperation with the MOVES Institute.

Approved for public release; distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2003	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: An Open Architecture for Defense Virtual Environment Training Systems			**5. FUNDING NUMBERS**
6. AUTHOR(S) Capt Stephen Matthews (USMC) and LT Kenneth Miller (USN)			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE** A

13. ABSTRACT *(maximum 200 words)*

This thesis develops a proposed software system that allows programmers to create virtual reality training environment applications for military (or other) use in which characters and character animation are necessary. Such applications are becoming more necessary to fill a gap in military training due to lack of personnel, time, money, and resources. Creation of virtual environment training applications allows military units to augment procedural training in preparation for live or physically simulated training. In the current environment of lesser training and more military requirements, such augmentation will only serve to benefit unit capabilities. While such systems for developing virtual environment applications are commercially available, those systems are costly in both licensing and usage fees. One of the tenets of the system that this thesis develops is that this system will be free and partially open source, such that programmers can create low cost virtual environment applications for military training, and such that experienced programmers can modify or add to the system in order to improve or enhance its capabilities to meet their needs.

14. SUBJECT TERMS Character Animation, Motion Capture, MOUT, Close Quarter Battle, CQBSim, Training, Virtual Environment, Open Source, Scene Graph, Software Architecture			**15. NUMBER OF PAGES** 197 **16. PRICE CODE**
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

AN OPEN ARCHITECTURE FOR DEFENSE VIRTUAL ENVIRONMENT TRAINING SYSTEMS

Stephen W. Matthews
Captain, United States Marine Corps
B.S., University of Illinois, 1997

Kenneth H. Miller
Lieutenant, United States Navy
B.S., University of North Florida, 1996

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2003

Authors: Stephen W. Matthews

Kenneth H. Miller

Approved by: Rudolph P. Darken
Thesis Advisor

Joseph A. Sullivan
Co-Advisor

Peter Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis develops a proposed software system that is usable by programmers to create virtual reality training environment applications for military (or other) use in which characters and character animation are necessary. Such applications are becoming more necessary to fill a gap in military training due to lack of personnel, time, money, and resources. Creation of virtual environment training applications allows military units to augment procedural training in preparation for live or physically simulated training. In the current environment of lesser training and more military requirements, such augmentation will only serve to benefit unit capabilities. While such systems for developing virtual environment applications are commercially available, those systems are costly in both licensing and usage fees. One of the tenets of the system that this thesis develops is that this system will be free and partially open source, such that programmers can create low cost virtual environment applications for military training, and such that experienced programmers can modify or add to the system in order to improve or enhance its capabilities to meet their needs.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

We would like to take this opportunity to thank several individuals who through their assistance, support, and advice have made the completion of this thesis possible. First, we want to thank Dr. Rudy Darken and CDR Joe Sullivan for having faith in our ideas and providing the resources, environment, and most importantly encouragement to complete our ambitious project. Erik Johnson and Matt Prichard combined their areas of expertise to enable us to put together a working system on several different platforms.

Finally we would like to thank our wives, Margie and Lana for their understanding and support during our time here at the Naval Postgraduate School. The completion of this thesis and earning our Masters Degrees would not have been possible without their support and sacrifice.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. PROBLEM STATEMENT

Military training is an absolute requirement, with more need now than ever before to ensure our service members are capable of performing the tasks and duties required of them in order to carry out the mission. Operational tempo has never been higher, with more forward deployed forces, more contingency planning, more international cooperative efforts, and more real world operations than before. In addition to operational tempo being so high, technology advances have provided the military with state-of-the-art equipment which has incredible capability but which also requires much more training than ever before. The ability to train underway and during the deployment cycle would allow greater flexibility in sustaining readiness over the deployment cycles, but currently, the tools to do so are not available.

In addition to the lack of deployable training tools, budgetary constraints are an additional burden on the ability to maintain readiness; training budgets are getting tighter, with training resources and facilities being either overtaxed or removed through base closures. Units needing to use a training facility have to schedule sometimes months ahead, and typically there are only a few training facilities aboard bases where there are many units in need of those facilities to get their personnel trained to required standards, often in advance of real world operations. Flight hours and ammunition, among a long list of expendable items needed for training, are expensive resources that the military is trying to find a way to cut the cost on, so that money can go to operations. The problem with cutting costs is that it also cuts the amount of training being conducted, forcing the military to either find more inexpensive means of training or do without.

B. MOTIVATION

The driving focus behind the application development that this thesis has inspired is that training budgets are getting tighter, time to train is becoming harder to come by, and there are times where we can take advantage of lost time with our service members, through virtual training environments, to ensure live training is as effective as possible. For example, when Marines go out as forward deployed units aboard ship in order to be capable of forced entry at a moment's notice, much of the time spent aboard ship is wasted with respect to training due to the inability to effectively train in the confined environment, and during that time, capabilities are doubly hurt by allowing skills to atrophy. While leaders try to keep their Marines' and Sailors' skills sharp through tactical decision games, map exercises, and other verbally conferred training, more effective use of time could be spent allowing small units to get some realistic, immersive, procedural and cognitive training, going step by step through standard operating procedures with small units in a virtual environment, with the capability of rehearsing until Marines and Sailors are well-versed and familiar with SOPs.

Virtual reality training tools are one method of enhancing live training and adding additional training time at a much lower cost. VR training environments remove many of the constraints of live training that make live training so costly; requirements such as travel, resource scheduling, and ammunition, supply, and equipment requirements are eliminated, with the only requirement for VR training being the computer hardware and equipment to run the software and the software itself. VR training can be used when resources are simply not available or when safety precludes the live training of standards that personnel are required to train to. An added benefit is that VR training can allow for the capture of data, which can aid in creating accurate and timely after-action reports.

C. HOW TO READ THIS THESIS

The rest of this chapter is divided into two major portions; the first provides an in-depth analysis on the use of virtual environments for training purposes, and the second concentrates on the development of software as a commodity, specifically with regard to scene graph engines. While these are necessary topics of understanding to any persons interested in the subject, they are background information and may be read independent of the rest of the paper. For those interested in delving directly into the open architecture for which this thesis was written, you will want to move directly to Chapter II. Chapter III is a system usability analysis that demonstrates a practical application created using libGF, the open architecture discussed in this thesis. The application not only demonstrates use of the architecture, but also provides experimental results on the differences in maneuverability between various input configurations. Chapter IV provides conclusions and recommendations regarding the use of an open architecture for creating virtual training environments. Chapter V concludes with future work that the authors see as useful to this thesis. Programmers who want to understand the basics of creating a libGF program and adding objects to the scene should review Appendix A.

D. USE OF VIRTUAL REALITY TRAINING ENVIRONMENTS (VRTEs)

Ten years ago, the average person could not tell you what virtual reality, cyberspace, or an artificial environment was. Today, virtual reality is a commonplace term to those in the computer field and is fast becoming an everyday term to all others. The reason for this phenomenon is that virtual reality tools and environments are better and more available, and businesses are seeing the training, promotional, and corporate potential for them. Militaries have also seen the capabilities of virtual reality environments and are currently developing and using this technology for recruitment promotion and training.

What militaries have to realize, however, is that the substance of the virtual reality simulations they use for training must meet criteria that ensure training is enhanced through proper use of training goals. Otherwise, virtual reality training environments may be nothing more than glorified, high-tech video games at best, and, at worst, can waste time that could be used on valuable training and can train personnel to poor or incorrect standards.

This section focuses on the potential and capabilities of Virtual Reality Training Environments (VRTEs) in current military applications, beginning with the requirements necessary of a VRTE to effectively train personnel, followed by the potential benefits and drawbacks of using VRTEs for military training. Next, several types of military training applications for which VRTEs are useful will be described, followed by specific examples of VRTEs being used or tested.

1. Requirements for Effective VRTEs

There are several requirements in order for Virtual Reality Training Environments (VRTEs) to be effective training tools. The leader wants to ensure that time is not wasted on ineffective training and that the VRTE being used is not training to poor or incorrect standards. Some requirements for VRTE training to be effective include:

a) Immersion

Being immersed, with relation to virtual environments, means being surrounded by something; everywhere you look, it's there. To create a sense of immersion in a virtual environment, we must be able to surround ourselves with various stimuli in a manner that makes sense and that follows rules similar to those of the real world. That is, when you turn your head to the left, you see the objects to the left of you. When you walk forward, you get closer to the objects in front of you. These are elementary features of our sense of being immersed in an environment; and when you're in a virtual environment, you expect the same results. (Aukstakalnis, 1992, p. 28)

For visual stimulus, a well-developed 3D viewing area is necessary to immerse the user in the virtual environment. For auditory stimulus, the ideal immersive auditory cues are those that sound to the user like they are coming from the location in the virtual environment that corresponds to its relative location in the real world with respect to the user's position. (i.e., a sound from five feet to the left of the user's position in the virtual environment sounds like it is really coming from a position located five feet to the left of the user) And for haptic (touch) stimulus, a truly immersive environment provides the user with haptic feedback when a part of the user's virtual self collides with or touches a part of the virtual environment. (i.e., when the user's virtual hand pushes on a table, the user feels the force applied as it would if he were doing the same action in the real world)

b) Presence

"When one is immersed in a synthetic environment and can interact with that environment one experiences presence. Presence is the experience of feeling that one is physically present within the computer-generated environment. Sometimes this experience is informally referred to as 'being there.'" (Johnson, 1997, p. 4) Concerning the need for and use of presence, the key is to distinguish the requirement for presence versus that for immersion. Immersion is when the user's expectations of the environment around him/her are correct; presence is the actual feeling of being present through not only immersion but through the ability to interact with the environment.

c) Specific Training Goals

For the VRTE to be an effective training tool, it must be geared toward specific training goals. Many current computer games provide enough immersion and presence to allow for a worthwhile training experience, but the reason they are video games and not training environments for military application is that the goal is to "win the game." Training environments, virtual or real, have training goals that must be accomplished in order to validate the participant's level of training.

d) Specific Virtual Tasks and Responses Geared Toward the Training Goals

For the VRTE to be effective, not only must it have specific training goals, but the virtual tasks and responses to the user's actions must all lead toward training the individual to the training goals; if the virtual environment does not keep the focus on the training goals, or at least constantly provide opportunity to focus on those goals, the user can wander in the virtual environment and thereby receive no training or ineffective training.

e) Realistic Scenarios that Play Out Realistically

Further, for the virtual tasks and responses of the environment to keep focus on the training goals, the environment must not lead the participant into "dead ends" or into an "end of the world" scenario. This is not to say that that virtual environments cannot have boundaries, but to be effective as a virtual environment, the participant should either not know that the boundaries are there or should know up front where the boundaries are and why.

f) Dynamic

The program or the instructor must have the ability to change the environment, in order to prevent stagnation in training. Where the video game again provides ample immersion and presence, what it does not normally provide is the ability for the environment to be different on every training exercise or, better, for the instructor to decide, exercise by exercise, what the environment will contain and how it will act and react.

g) Scalability

The program or the instructor must also have the ability to scale the environment up or down to meet the size and experience of the unit being trained and to provide the level of training commensurate with the unit size of the trainees. In addition, if the need is present to train across units, the environment needs to be able to handle training across units through interoperability of systems.

h) Feedback

Finally, the environment must provide feedback to the instructor and to the participants in order to validate whether the training goals are being met, how well, and, if not, what the participants need to change or perform better to reach the training goals. Without feedback, the effectiveness of the application as a training tool is diminished by the effect that a) the participant will only learn what he can see or what he does himself (as opposed to learning from a general overview of what happened or learning from other participants' actions), and b) the ability of an instructor to evaluate the participants' actions and decisions is diminished.

2. Benefits of Using VRTEs

There are many benefits to using Virtual Reality Training Environments. (VRTEs) No VRTE can replace full-scale, live training, but using them to supplement training that is costly and therefore not performed as often as necessary allows users to gain cognitive and procedural skills that will enhance the live training that is conducted or, in the absence of live training, will give the participants at least a solid foundation of cognitive and procedural skills from which to draw from when faced with a real scenario. Some of the specific benefits of using VRTEs include:

a) Training to Specific, Focused Goals

VRTEs, being computer simulations, can be designed to lead users toward goals so that training time is well spent and so that the user remains focused on the training goals without explicitly realizing that the simulator is leading him/her toward those training goals.

b) Repetition of Same Training

VRTEs allow users to repeat skills as many times as needed or wanted and as often as necessary in order to ensure that the user becomes proficient in the rudimentary skills being taught; the saying "practice makes perfect" holds well here, as the user is capable of practicing the same task over and over until he/she can perform well. This is not always the case in live training where there may be a time or cost factor associated with the training that prevents practicing skills to proficiency.

c) Potential for Comprehensive, Objective Review and After Action Reports (AARs)

The parachute trainer fielded by Systems Technology, Inc., provides solid example of this benefit. "The instructor may freeze the simulation at any time to discuss the progress of a jump, and then may continue or end the jump. The instructor records all jumps for instructor and trainee review using a playback option. The playback shows the motion as viewed by the trainee and includes a display of the trainee's toggle inputs. The simulator evaluates completed landings. These evaluations are displayed on the instructor's screen and can be printed to provide a report of a trainee's progress." (Hogue, 2003) The potential for training benefits through comprehensive AARs is only limited by the capability of the VRTE being used.

d) The Ability to Add or Remove "Stress Levels" or Battlefield Characteristics as Needed

VRTEs allow for setting the optimal level of training for the participants, as well as adding or removing opposing forces or background effects. This allows for training (and thereby gaining a performance edge) in many different configurations of an environment, a capability that most live training cannot provide in a condensed period of time. Dr Sarkar and Terrance Tierney write of weapons system testing in VRTEs, "Battlefields are chaotic environments. Fighters often win when they are able to take advantage of the environment and mobilize them in their favor. However, achieving a performance edge under each particular environmental state is difficult due to finding or recreating specific location, terrain, weather, and enemy/friendly force mix. Yet, there has not been a cost-effective means available to change characteristics of terrain or bring together a particular battlefield scenario. With the capability of virtually modifying a replica of a real terrain under a particular condition with a particular force ratio with predetermined performance abilities, one can test various conjectures, leading to a vastly optimized weapon system. […] The synergy of simulation and virtual world has opened an entirely new dimension for optimizing weapon systems." (Landauer, 1998, p 131) The quote is easily extended to encompass not only weapons systems but also overall training of forces.

e) Provides for Unsafe or Potentially Unsafe Training to be Conducted in a Safe Environment

VRTEs allow training to unsafe levels, allowing participants to gain valuable skills they cannot train to in a live environment where the same training would be fatal or unsafe. Examples include:

Nuclear, Biological, and Chemical (NBC) training – training to standards of response time in harmful environments, based on cues from the virtual environment

Firefighting skills – virtual close quarters fire fighting environments can be created allowing military firefighters and personnel with supplemental tasks of fire fighting to conduct realistic, procedural training in a safe environment

f) Cost Savings

The use of VRTEs has the potential to substantially reduce training costs overall by making users more proficient prior to live training exercises, which then require less time and evolutions. Examples include:

- Indoor Simulated Marksmanship Trainer (ISMT) – use of the ISMT reduces rifle range requirements, the number of re-qualifications, and the number of rounds needed to train Marines to standards of marksmanship

- Flight trainers – the use of flight trainers reduces the number of flight hours needed in the air and enhances those hours spent by pilots in the air by providing needed reinforcement of basic procedures and scenario responses
- Tank trainer – From March 1993 to 01 July, 2002, the Ft. Knox User has logged 1,087,323 simulated miles driven by 36,198 M1 driver trainees and 2,878 M1A2 advanced driver trainees. The typical Training Tank costs $155/mile to operate whereas the Tank Driver Trainers cost $5.44/simulated mile. Based on these figures, the cost of operating the 20 trainers has been $5,915,037. The total cost avoidance by training with the twenty TDT's is $168,620,028 over the past 9 years and 4 months. The cost avoidance is 285% of the total project development cost of $57M. (Tank Driver Trainers, 2003)

g) Reduced Setup Time

In comparison with live training exercises of the same size as the VRTE exercise, exercise preparation and setup time can be reduced, allowing more time to be spent on the training skills or on other missions.

h) Small Footprint

In comparison to the size of the area required for a training exercise and all associated equipment, many VRTEs (particularly those that are PC-based, standalone systems) have the advantage of taking much less space than the representative live training.

For example, the Air Traffic Control Virtual Reality (ATCVR) air traffic control system requires no more than a 10' x 10' x 8' area to conduct simulated tower operations with as many as 40 aircraft in virtual patterns. The Fire Team Cognitive Trainer (FTCT) for the Deployable Virtual Training Environment (DVTE) is designed to be deployed for squad tactical training while aboard ship, using no more than 16 laptop computers that, since they are networked, can be used in whatever space is advantageous for the squad, platoon commander, and opposing forces to occupy. (see Sect. I.D.5 for reference to both examples)

i) Large Scale Virtual Networked Training Environments

VRTEs, if designed as such, allow for a large number of participants and participation in the same training exercise by personnel in different locations. A practical example of large scale, networked VRTE use could be emergency services training: “By the use of computer networks, it would even be possible for military and civilian emergency agencies to ‘virtually’ work and drill together, as might prove necessary in the event of a terrorist attack, chemical/biological release, or other large scale disaster situation. ‘The possibilities of this concept are endless...for improving our national response capabilities and facilitating effective and more realistic training for those that protect us on a daily basis.’" (Anderson, 1996)

j) Provides an Experience Level that can Potentially Accomplish the Mission and Save Lives

All other benefits aside, the ultimate goal of training is to ensure mission accomplishment. Additionally, in the process, preventing the loss of life is also an important result. VRTEs can provide needed training that might otherwise not be accomplished, leaving service members lacking in skills needed in a real engagement/scenario; as such, the knowledge and skills one has gained by VRTE training is directly valuable in mission accomplishment and in preventing loss of life. "One of the biggest problems in both the military and emergency services is to expose 'rookie' responders or low-ranking soldiers/sailors/airmen to enough 'experiences' to allow them to function effectively in a crisis or combat situation. Historically, many injuries and even deaths occur in both training and in their first few months or even years on the job. It is believed by [Emergency Response & Research Institute (ERRI)] and others that by using realistic virtual reality training that this 'experience quotient' can be rapidly increased and unnecessary casualties prevented." (Anderson, 1996)

3. Drawbacks of Using VRTEs

There are also many potential drawbacks to using Virtual Reality Training Environments. (VRTEs) While VRTEs can provide positive supplemental training to a good training program, there are potential drawbacks that can seriously detract from the benefit gained by using VRTEs, and can, in fact be detrimental to the training program in a broader sense than simply not providing solid training. All of the following drawbacks are recognizable and can be either engineered out of the system or educated against; if they or other drawbacks are present and not addressed, training with VRTEs will not prove effective.

a) Lack of Realism

For VRTEs to be effective training tools, they must provide a sense of realism, so that what the participant sees in the virtual world can be translated to training and missions in the real world. VRTEs can lack realism for several reasons, as listed below; the reasons listed are not exclusive, nor are they always applicable. (i.e, if training goals require a visual familiarization of terrain, haptic feedback and other cues, such as auditory cues, may not be needed and, as such, will not so greatly contribute to the lack of realism)

- <u>Lack of realistic cues</u> -- if realistic cues (correct visual proportionality, spatially oriented sound as needed, etc) are not provided, the user does not feel immersed in the VRTE.
- <u>Limited field of view</u> -- the visual display is the user's window into the virtual world; if it is limited, the user may not experience the immersion needed to train effectively in the VRTE.
- <u>Lack of realistic (if any) haptic feedback where needed</u> -- adding to the lack of realistic cues, haptic feedback can determine whether a user feels present, so, where needed, its lack could add dramatically to the lack of realism.
- <u>Limited range of motion or travel</u> -- participants may be limited in the virtual world from moving their head and limbs as needed or may be limited in the locations they can move to.
- <u>Levels of physical disability can be skewed in the VRTE</u> -- training environments can have a very “game” flavor to them in that there are multiple lives, multiple hits, or other unrealistic skews that are incorrect characteristics and create a lack of realism.

b) Requirement to Learn How to Use the System and Equipment

Learning to use the system and equipment can be time-consuming, cumbersome, and frustrating. Training that frustrates the participant is often ineffective training, no matter how well the training was oriented to the goals. In addition, time outside of the training evolution itself needs to be spent learning the system and equipment prior to the training evolution commencing in order to receive effective training; otherwise, the trainee is just spending time learning the system. The participant must become familiar with the input setup for devices such as keyboard, joystick, or any other non-intuitive input device and become orientated with the information being provided, such as HMD display or screen display.

c) VRTEs can Train Poor or Incorrect Practices

Training button pushes or other non-realistic inputs in place of live actions that would be executed in the real world creates the false impression of how to perform the given task in the real world--the VRTE can train users, incorrectly, to push a button in a live environment where a real action is necessary. For example, if a user is immersed in a VRTE using a head-mounted display, yet uses a button to aim in with the sights of a weapon, that same participant may hesitate momentarily in sighting in his weapon in a live environment, as he is mentally seeking the same button press. While it is true that VRTEs can train poor or incorrect practices, it needs to be stated that live training exercises have as much potential; i.e., teaching medical personnel how to deal with "cherry pickers" (simulated casualties) in any way other than how they would deal with real casualties promotes incorrect practices that could cost lives. This drawback is present and must be guarded against in all training environments, not only in VRTEs.

d) VRTEs can Neglect Procedures that Lead to Lack of Ability

Not only can VRTEs train poor or incorrect procedures, but--just as detrimental--VRTEs can also neglect to train necessary steps or tasks required to accomplish a mission. For example, if the VRTE is training flight runup procedures that leave out a step that would be required in the real world, the pilot is taught to neglect that step. Neglecting steps in training causes gaps in task performance during mission execution that can cause mission failure.

e) Frustration can Lead to Lack of Desire to Train

VRTEs that are hard to use and navigate create frustration for the user. Frustration with a training environment, virtual or otherwise, erodes morale and leads to a lack of desire to train. Even if the application is easy to use, if the user does not feel an adequate sense of immersion and/or presence, the user will "feel" as though there is something missing and will become frustrated from trying to work in an environment that he/she does not feel a part of.

f) Negative Impacts

There are several distinct effects of VRTEs that can have negative impacts on the participant as well as on training. These include:

- <u>Cybersickness/Simulator sickness</u> -- A study conducted on the effects of and propensity for simulator sickness in virtual training environments reports, "There was an inverse relationship between simulator sickness and amount learned in the synthetic environment. Soldiers who reported greater levels of discomfort tended to learn less […] than their peers who experienced less simulator sickness." (Johnson, 1997, p. 52)
- <u>Neck injury</u> -- specifically related to HMD use or use of any other head-mounted equipment, neck injuries can occur when participants wear excessively heavy headgear or headgear that is not balanced (pulls the head down in one direction) for lengths of time or while exerting the neck muscles.
- <u>Lack of physical exertion gained from live training</u> -- Some live training exercises aim at the additional goal of physical fitness through strength or endurance training. Extensive use of VRTEs that do not provide the same physical exercise that their live-training counterparts provide can lead to erosion of the level of physical fitness that personnel would otherwise maintain.

The negative impact that these effects can have creates frustration and a lack of desire to train, which is overall detrimental to mission accomplishment.

4. Prospective Military Training Applications for VRTEs

"One of the earliest uses of simulators in a military environment was the flight trainers built by the Link Company in the late 1920's and 1930's. These trainers looked like sawed-off coffins mounted on a pedestal, and were used to teach instrument flying. The darkness inside the trainer cockpit, the realistic readings on the instrument panel, and the motion of the trainer on the pedestal combined to produce a sensation similar to actually flying on instruments at night. The Link trainers were very effective tools for their intended purpose, teaching thousands of pilots the night flying skills they needed before and during World War II." (Baumann, 1993)

What we have to remember when considering whether military training exercises make suitable platforms for virtual reality applications is to look at the goals of the training and see whether or not we can achieve those training goals correctly through virtual reality. If the answer is 'no', then to create a VRTE for a set of training goals we cannot hope to reach through virtual tools creates nothing more than a meaningless video game at best, and at worst, detriments unit training. If the answer is 'yes', then we have the potential to supplement training with VRTE technology and reward the military with better cognition, better procedural knowledge, or better decision-making skills.

The following are applications where VRTEs can, and in some cases do, supplement training to achieve training goals:

a) Equipment Familiarity or Usage Procedures

VRTEs can familiarize personnel with equipment or teach them how to use that equipment prior to ever using the real equipment. This promotes safety, as personnel will already be familiar with safety features and procedures prior to use of the equipment. It also enhances training and mission accomplishment by giving personnel additional experience that they would otherwise not receive. Some examples of equipment that can be created in VRTEs for personnel usage and familiarization include:

- <u>Aircraft</u> -- fixed wing and rotary wing

- Vehicles -- The 7 1/2 ton Medium Tactical Vehicle Replacement (MTVR) has a simulation designed for driver training; Engineer Heavy Equipment (HE) can be simulated to familiarize or train the user

- Utility equipment -- generators and shipboard equipment

- Weapons -- small arms and systems

- Night vision equipment -- head mounted and weapons mounted

b) Procedural learning of Standard Operating Procedures (SOPs)

VRTEs are capable of training SOPs by requiring step-by-step procedures to be performed in order to accomplish the mission. Examples include learning rules of engagement (ROE), combat SOPs, radio procedures, and small unit tactics. The advantage to using VRTEs for this procedural type learning is the ability to train as many times as is needed to gain proficiency, and to be able to stop or rewind the scenario to get a macro look at what is being done correctly or incorrectly.

c) Navigation Skills

VRTEs can assist in learning dead reckoning skills or equipment (i.e., compass) related navigational skills. VRTEs can provide an overhead view of the correct path and of the path taken by the user, helping him to understand where he made mistakes. They can also provide current hints to the user while navigating in the virtual environment, speeding the learning process by allowing the user to learn as he moves through the environment, not just afterward. The computer environment of a VRTE is unfailing in its ability to determine what path should have been taken and what path was taken, making it a useful learning tool. In addition, navigation can be performed on virtual non-flat terrain, with visual cues and hints, to show the user what a straight compass heading over terrain really looks like.

d) Terrain Appreciation/Environmental Familiarization

VRTEs are also useful in learning terrain features in general, from a visual standpoint. What is unique about using VRTEs for viewing terrain features is that the user can detach himself from the environment and "fly" around, viewing a terrain feature from every angle in order to better understand and appreciate the terrain. This type of training lends itself to training for real missions; VRTEs can be used as virtual sand tables for preparing for a training exercise or a mission; they can be used to provide commanders with reconnaissance of an area so that he can direct training or mission; and they can be used to provide intelligence analysts with information with which to train and educate the unit on the layout of areas, countries, regions, etc.

e) Decision Making Skills Training

VRTEs can be used to provide a realistic environment and scenario designed to train personnel in decision-making skills. Scenarios can be designed where the leader must act in a timely manner while the environment and scenario continues to change, and where the decisions made within the virtual environment affect the environment and the course of actions. Such decision making training environments can benefit individual cognition, judgment, and leadership capabilities.

5. Examples of Current VRTE Military Applications

There are many examples of current applications of virtual training environments being used or developed. For reasons stated already, the subject has proven to be the focus of several efforts to introduce more training, lower-cost training, and training that makes real operations safer and more effective. The following list of examples is far from complete, but shows some of the implementation and research currently or recently conducted.

a) Combined Arms Tactical Trainer (CATT)

The US Army's simulation, training and instrumentation command (STRICOM) has been developing simulation systems for years. SIMNET, the Army’s simulation networked environment brought together many individual simulations over great distance to provide a virtual battlefield in which participants could actively train with other participants, and in which commanders could lead large-scale battlefield units. The successor to SIMNET is the Combined Arms Tactical Trainer (CATT) family of simulation trainers. These include trainers in the areas of ground, air, command, and support. The prime contractor the Army uses for the CATT family of simulators is Lockheed Martin.

Figure 1. Combined Arms Tactical Trainer (CATT) (From Ref.[1])

The following are examples of CATT modules:

- **Close Combat Tactical Trainer (CCTT)**

The Close Combat Tactical Trainer (CCTT) system is the centerpiece of the CATT family. It is a fully distributed interactive simulation system and consists of a networked array of vehicle trainers. The vehicle trainers are members of the Ground or Air Combat Tactical Trainer families of simulators. The compilation of simulators into a single distributed system produces a synthetic battlefield on which participants can do unit training. Semi-automated forces can be added into the synthetic battlefield as aggressors or friendly units.

[1] http://www-leav.army.mil/nsc/tsm/ and http://www.ets-news.com/catt1.html

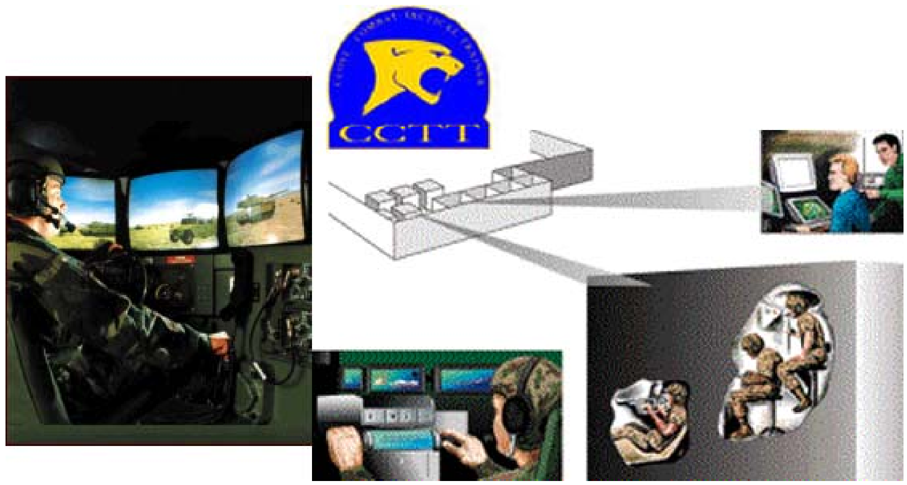


Figure 2. Close Combat Tactical Trainer (CCTT) (From Ref.[2])

- **The Family of Ground Combat Tactical Trainers (GCTT)**

The Ground Combat Tactical Trainers are a family of virtual trainers covering Armor, Infantry, Field Artillery, Air Defense and Combat Engineer systems. GCTT trainers includes conduct of fire trainers, driver trainers, and maintenance trainers, as well as any other trainers needed for ground combat systems. These systems are standalone virtual training systems, though some are capable of being integrated into the CCTT and CATT environments. Currently, GCTT includes the following systems:

  - *Tank Driver Trainers (M1 and M1A2 TDT)*
  - *Advanced Gunnery Training System (AGTS)*
  - *Abrams Full Crew Interactive Simulation Trainer XXI (AFIST XXI)*
  - *M1/M1A2/M1A2 SEP Tank Driver Trainers (TDT)*
  - *M1A2/SEP Maintenance Training System (MTS)*
  - *M2A3 Bradley Maintenance Training System (MTS)*
  - *Basic Electronic Maintenance Trainer (BEMT)*

---

2 http://orlando.drc.com/Whats_New.htm, http://www.amso.army.mil/smart/documents/ref-guide/sec-VI/tools.htm, and http://www.ets-news.com/virtual.htm,

- *Multiple Launch Rocket System Maintenance Trainer System (MLRS MTS)*
- *Fire Support Combined Arms Tactical Trainer (FSCATT)*
- *Guard Unit Armory Device Full-Crew Interactive Simulation Trainer (GUARDFIST II)*
- *Forward Observer Exercise Simulation (FOXS)*
- *Stryker Training Devices*
- *Engagement Skills Trainer 2000 (EST 2000)*
- *Wolverine Driver Mission Trainer (DMT)*
- *Linebacker Table Top Trainer*
- *Avenger Table Top Trainer*
- *Advanced Concept Research Tools (ACRT)*
- *Recognition Of Combat Vehicles (ROCV)*

b) Deployable Virtual Training Environment by Coalescent Technologies Corporation

The Deployable Virtual Training Environment (DVTE) is a distributed interactive system designed to provide a tactical 3D simulation environment for cognitive, procedural, and tactical training at the individual and unit level.

"Distributed Interactive Simulation is being developed for the Marine Air/Ground Task Force (MAGTF) which specifically includes Air Support, Light Armored Vehicles, Infantry, and Forward Observers with a constructive virtual simulation using the MAGTF Federated Object Model (FOM) and High Level Architecture (HLA)." (Deployable, 2003)

The Deployable Virtual Training Environment (DVTE) allows for tactical and cognitive training in an interactive virtual environment where each entity connected to the environment can see and interact with each other entity; i.e., the forward observer will be able to interact with artillery and air support components to bring fires to the target; dismounted infantry will be able to interact with Amphibious Assault Vehicle (AAV) support to reach the beachhead or press forward from the beachhead.

The distributed interactivity of the system makes it an excellent means to provide combined arms training and, through the use of High Level Architecture (HLA), allows the system to connect with other military simulations, creating an environment for joint service training opportunities in a virtual environment.

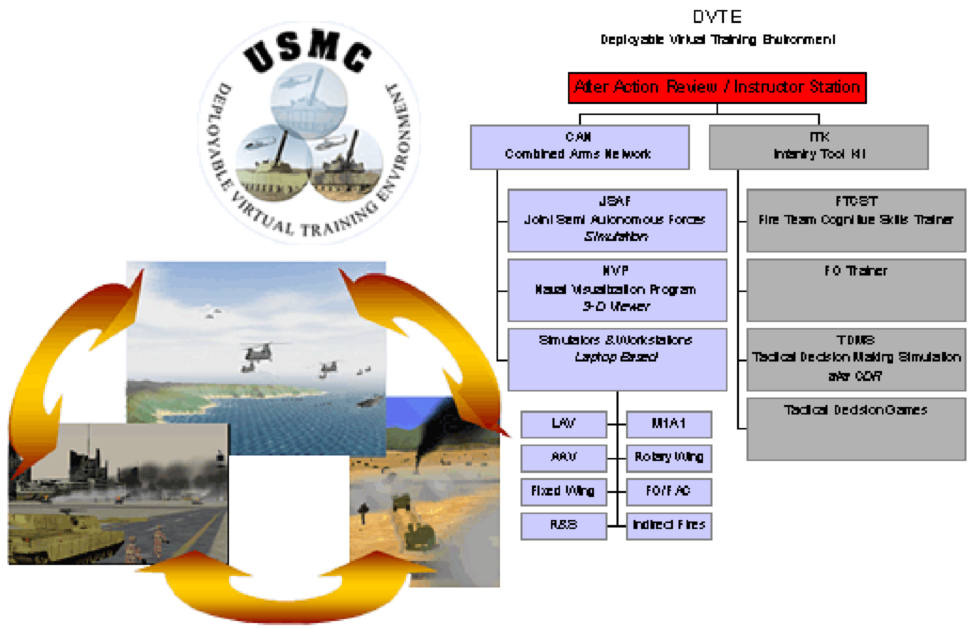


Figure 3. Deployable Virtual Training Environment (DVTE) (From Ref.[3])

- **Fire Team Cognitive Trainer**

The Fire Team Cognitive Trainer(FTCT) is a Virtual Battlefield System-1™ component to DVTE (VBS1™ is created by Bohemia Interactive Studio). FTCT trains Marines, particularly small unit leaders, tactical judgment, situational awareness, and decision-making skills, in order to improve small unit effectiveness on the battlefield. FTCT is fully interactive in a 3D training environment, and allows the participant flexibility of action. Small units can also practice skills and gain or recall procedural

[3] http://www.ctcorp.com/performance15.html and http://www.tecom.usmc.mil/techdiv/dvtepaper3.htm

knowledge through tactical simulation. FTCT has the ability to provide aggressor entities, in either computer controlled or human participant controlled form.

FTCT is a Virtual Battlefield System-1™ simulation (VBS1™ is created by Bohemia Interactive Studio). VBS1™ is a simulation engine which has the dynamic capability of being able to load user-defined terrain and surroundings, and the flexibility of being able to change the environment and its conditions at any time.

> VBS1™ offers realistic battlefield simulations and the ability to operate a myriad of land, sea and air vehicles across vast outdoor terrains. Instructors may create virtually any imaginable combat scenario and then engage the simulation from multiple viewpoints. The advanced squad-management system enables participants to efficiently issue orders to squad members as well as coordinate assaults. VBS1™ allows free-play within scenario-based training missions. It also incorporates real-time simulation of wind, rain, fog, clouds, time-of-day, sunrise and sunset, and tides.
>
> VBS1™ may be used to teach doctrine, tactics, techniques, and procedures of squad and platoon offensive, defensive, and patrolling operations. VBS1™ delivers a synthetic environment for the practical exercise in the application of the leadership and organizational behavior skills required for the successful execution of small dismounted infantry unit missions. (Tactical, 2003)

Figure 4. Fire Team Cognitive Trainer (FTCT) (From Ref.[4])

---

4 http://www.flashpoint1985.com/press/vbs1.html and http://www.tecom.usmc.mil/techdiv/ITK/VBS1/VBS1Draft.htm

c) Shiphandling Simulation Training by Marine Safety International

Marine Safety International (MSI) provides shiphandling training to U.S. Navy personnel at MSI Newport, Rhode Island, MSI Norfolk, Virginia, and MSI San Diego, California. Training is completed to real-world standards on tasks such as mooring, anchoring, buoy mooring, underway replenishment, man-overboard, and emergency procedures. Training is accomplished at a procedural level of learning the skills necessary to command or pilot a ship through various circumstances. While there are ten standard training exercises that a ship can fashion training around for its personnel, the shiphandling trainer is versatile and flexible in its training capabilities.

Training is accomplished in a realistic shipboard environment with all ship hardware looking and acting as it would aboard a real vessel. This provides the level of immersion and presence necessary for personnel to get realistic training on full real-world tasks, without the loss of training as to which shipboard components are needed for each task. Participants are provided a full panoramic view from inside the bridge and a partial panoramic view from the bridge wing.

> The simulators include the ability to train the full bridge and CIC/CDC team including lookouts, bearing takers, helmsman and navigators as well as the OOD and conning officer. Shiphandling training at MSI Newport consists of a set curriculum of basic through advanced evolutions with the primary focus on mooring and unmooring from a pier, underway replenishment and emergency shiphandling. (Shiphandling)

Figure 5. MSI Shiphandling Simulator (From Ref.[5])

[5] http://www.marinesafety.com/usn.html

d) Air Traffic Control Virtual Reality (ATCVR) Training System {delivered by Southwest Research Institute (SwRI)}

Air traffic controller certification requires extended intense training. Because ensuring the safety of all persons is such a critical factor to air traffic control personnel, many hours of training and qualification are necessary to ensure that air traffic safety is upheld. The Air Force has seen the need for virtual training environment capabilities in order to certify personnel and maintain currency in air traffic control procedures proficiency. The Air Education Training Command (AETC) is conducting studies in evaluating the Air Traffic Control Virtual Reality (ATCVR) for just that purpose. The ATCVR is a development of Southwest Research Institute (SwRI) and is currently being tested at Randolph, Luke, Vance, and Altus Air Force Bases.

ATCVR is an important product to the training of air traffic control personnel because it provides more hours of training in a safe environment, with room for errors and inspection of those errors during after-action reviews, prior to personnel becoming responsible for the safety of real lives and aircraft. Whereas live training of personnel is time, resource, and man-hour intensive, ATCVR training requires two personal computers, a head mounted display, and an instructor in order to conduct individual training. ATCVR provides the flexibility to train personnel at any time and on any conditions, allowing the repetition of training on conditions that personnel find stressful or difficult.

> The goal of the ATCVR simulator is to familiarize a trainee with the local airfield environment, traffic patterns, procedures and radio calls used in the tower. The system accomplishes this by simulating base-specific traffic patterns using instructor-selected flight plans and aircraft in a simulated environment closely matching the airfield. The ATCVR system provides a 360-degree view that immerses the trainee in a realistic tower environment allowing the trainee to become familiar with the local tower prior to working with live traffic. (Fisher, 2001)

Figure 6. Air Traffic Control Virtual Reality (ATCVR) (From Ref.[6])

e) Helicopter Navigation Studies at the Naval Postgraduate School

Helicopter Terrain Navigation Training Using A Wide Field Of View Desktop Virtual Environment, by CDR Joseph A. Sullivan (USN) – "Helicopter terrain navigation is a unique task -- training for this task presents unique challenges. Current training methods rely on dated technology and inadequately prepare pilots for real-world missions. Improved training specifically tailored to address the unique needs of the helicopter community that capitalizes on recent improvements in desktop virtual environment (VE) technology could substantially improve the training process and reduce training costs."

An Interactive Virtual Environment For Training Map-Reading Skill In Helicopter Pilots by Capt Tim McClean (USMC) -- Currently, Student Naval Aviators are trained to interpret 1:50,000 scale contour maps by watching VHS videotapes. These tapes show a helicopter moving about twice its normal speed over desert terrain. The helicopter does not stop until the tape is over, hence, the training evolution quickly becomes useless because students usually make mistakes during the first minute of the tape and are unable to recover or to learn from those mistakes. Based on a previous study

[6] http://www.tspi.swri.org/pub/2001iats_atcvr.htm

at the Naval Postgraduate School, a training system that utilizes virtual environment technology was developed. This desktop system was fielded at Helicopter Antisubmarine Squadron 10 (HS-10) for experimentation. Results of this experiment indicate that student pilots who received VE training performed the navigation task better in the helicopter than students who received only conventional training. Also, an IT-21 Wintel based computer is capable of rendering a graphically intensive multi-monitor application at frame rates suitable for training."

Exploring a Chromakeyed Augmented Virtual Environment (ChrAVE) for Viability as an Embedded Training System for Military Helicopters by Maj Mark Lennerton (USMC) – "The ChrAVE mixed reality helicopter training environment was created in an effort to provide an immersive and familiar environment for training pilots embarked aboard ship during deployment. The system is designed to exercise pilot skills as rigorously as real flight operations would, during periods when live flight hours are not available. Computer-generated terrain provides the pilot's view of the outside, while a camera-generated view of the cockpit simulation provides the pilot's view of the cockpit interior and instrumentation." (Sullivan, 2003)

E. DEVELOPMENT OF SOFTWARE AS A COMMODITY

The focus will now shift from reasons for using VR technology in military applications to methods of developing these VR training capabilities at lower cost. Hardware has already become a commoditized item, in which the user can feel relatively comfortable choosing whatever brand of hardware he likes (and that can be bought at the best price); the customer knows that, across the spectrum, a hardware item from Brand A will perform the same as a similar hardware item from any Brand B. This commoditization of hardware has a) lowered the general cost of hardware as a commodity, and b) created a common functionality expected and delivered from similar hardware.

Software has, for some time, been at a stage where capability was based on the authoring company of the software; different companies would write dramatically different software, regardless of the similarities in application use and need. Until recently, standardization of software appears to have been regarded as unnecessary. However, the population of computer users has dramatically increased over the last ten years, and those who are using computer applications tend to know what they want and what they want it to look like. The average household user knows how to send mail using Microsoft Outlook®[7] and does not even want to look at anything that appears different.

Because this user opinion is becoming more of a force in the development of new software in areas that are already well developed enough to elicit user opinion, software is now at a point where functionality is becoming more standardized across similar software products. Many software concepts have already been tackled and effectively implemented, allowing for the creation of software patterns. These patterns find themselves in the form of reusable building blocks that can be put together to create generic architecture styles that are optimized with regard to functionality and reusability. What this effectively creates is the ability to commoditize software, much like was done with hardware.

In the field of virtual reality simulation engine software, the user—in this case, the application programmer—has generally seen or worked with several packages, and has seen similarities between them. Many new simulation engines are being developed, and these are incorporating a standardized set of functionality. The differences among the functionalities of these simulation engines is becoming smaller, making cost and performance the driving factors behind use of one engine over another.

1. Simulation Engines

a) Basic Overview

Virtual reality applications can be extremely complex to write as independent applications; many different functionalities are needed in order to provide a full, rich set of features that capitalize on the latest developments in hardware to provide the user with as immersive and realistic of an environment as possible. Writing a single

[7] Microsoft Outlook® is a registered trademark of the Microsoft Corporation; http://www.microsoft.com.

application that encompasses all of those functionalities would be time and resource intensive, and would violate the principles of code reuse, flexibility, and extensibility which allow programming projects the ability to adapt to a particular need and to save time through the use of existing resources.

A better solution is to bring together a compilation of all needed functionalities, taking advantage of encapsulated, reusable code in order to create a modular, extensible architecture that is as rich in capability as possible while also being as optimized in performance as possible. While a simulation engine necessarily has to be general enough in architecture to accommodate a wide variety, size, and scope of applications, because optimal code is implemented into the engine, it sacrifices little optimization. The end product is a software architecture that is optimized, scalable to all applications it is designed to accommodate, and rich in functionality.

b) Commonalities List

- Scene graph

At the heart of every virtual reality simulation engine, whether its purpose is to create simulations, training applications, or games, is a scene graph that takes care of the basic functions needed to take the application state and create a rasterized picture on a viewing screen. A scene graph, in keeping with its name, is an acyclic, directed graph, or a 'tree'. The graph has a root node that contains nodes representing the 3D visible objects, as well as all data needed to render the 3D scene, such as lights, textures, and states. A scene graph takes the place of voluminous code, which would otherwise need to be written for every application to draw the scene at every frame. The scene graph follows at least the following three basic principles of functionality.

The first principle that the scene graph follows is that it optimizes scene rendering. A scene rendering program that renders every element of the scene on every frame is extremely inefficient. The scene comprises all elements that can be drawn or the information to influence those items being drawn. Much of the scene is not within the viewing frustum at any one point in time; the scene graph takes advantage of this fact by determining which elements are not in the viewing frustum and culling those items, preventing the rendering pipeline from having to waste cycles determining how to render them when they will not be rendered anyway. Additionally, of those items within the visible frustum, some are at such a perspective virtual distance from the observer's viewpoint as to show very little visual detail. The scene graph takes advantage of both of these points.

The second principle of scene graphs is portability. It is no simple task for the programmer writing the individual application to bring together all of the tools needed to make a scene rendering program portable across various platforms. The scene graph, however, can be written to take advantage of—possibly at runtime—various packages written to allow for cross-platform capabilities.

The third principle that scene graphs adhere to is scalability. An application that contains its own scene rendering functionality will generally be scaled specifically for that application; it would be expensive, in terms of resources, to write every application with the ability to handle an arbitrary number of entities. This is where an individual application which includes its own rendering code can be more optimally programmed than if the application made use of a scene graph; however, the tradeoff is that using a scene graph, which is written generically for an arbitrary number of entities, includes the optimizations written into a scene graph which would otherwise be time-consuming to implement into individual applications.

The scene graph is the foundation component to the simulation engine, and it provides object rendering and manipulation capabilities basic to the generation of a visible scene. Objects are made of pieces of geometry, which are made of triangle faces, which contain vertices. The scene graph contains and keeps track of all of the pieces of those objects so that they can be efficiently rendered in the scene as the application needs.

- Effects system

In general virtual reality applications use effects as a means to create a realistic, immersive environment; eye candy equates to realism, and, so, becomes an important factor to the application. Environmental conditions such as fog, rain, wind, sun make applications seem more believable and enhance the user's experience accordingly.

Particle generation is an import capability of an effects system in a simulation engine, as it can be used for a diverse array of applications, such as smoke, water spray, or engine exhaust. The effects system in a simulation engine can effectively convey a sense of realism to the application end-user.

- Physics engine

If a simulation involves realistic physical movement of objects in the scene, then a physics API will need to be incorporated into the engine. In the absence of physics computation, objects move in a sterile method about the scene, moving discretely in the direction they are told and for as far as they are expected. Providing visual cues that objects have real, physical properties involves the physical interaction between objects, such as a ball that rolls off of a table and bounces on the floor. To provide this interaction, computation needs to be done to account for where objects are in the world and what forces are acting on them at any given time.

- Character animation

The addition of character animation to an application adds more than what seems apparent. The geometry of a model is maintained and rendered by the scene graph, but many other components of an animated character are not. The animation of characters often involves key framing, which is a means by which, in local reference, the vertices of the model move from one frame to the next in order to produce local movement of part or all of the model. This differs from global movement of an object in the scene in that objects moving in the scene maintain the same local shape—all vertices are moved as one. Character animation essentially involves deformation of the model from one state to another, over a series of steps (key frames). For efficiency, these key frames do not generally maintain the state of every vertex in every key frame; that would be resource intensive. Character animation APIs find a way to store this information efficiently, such as by maintaining only the change in vertices from one key frame to the next, which allows for the efficient rendering of objects in the scene that can not only move around the scene, but can be moving locally as well; this allows for the creation of human characters, for instance, who not only move around the scene (stiff, as though figurines), but can move (or deform) as needed to look as though they are performing actions such as walking, running, or waving.

- Game networking

Simulations do not generally involve single entities moving around a scene by themselves; training simulations that are made to train small units, for example, need to be able to put the entire unit in the simulation at one time. This involves a networking capability so that each entity can move around the scene and interact in whatever manner necessary with all other entities.

- Artificial Intelligence for agents

When more participants are needed in a simulation than are available, or when the hardware does not support as many participants in an application as are needed, additional participants can often be created through the use of intelligent agents, or bots. Bots can provide opposing force simulation capabilities if the human participants in a simulation are to all work together as a unit; training a small unit to react to an ambush does not necessarily require that real persons simulate the opposing force lying in wait, as long as that opposing force can behave in an intelligent manner expected of them. Often there can be simple or repetitive tasks that provide no training value but need to be performed by an entity in the application; agents can perform those tasks. This serves to provide needed actions or interactions in an application without the expense of human resources.

- Sound, including spatial/positional sound

Sound in virtual reality applications can be a key factor to immersion; sound can also provide information in a virtual environment that would be difficult to receive by other means. Additionally, correct training transfer of procedures and techniques may involve sound, such as training applications in which air traffic controllers listen to pattern aircraft traffic over radio. The spatialization of some sounds is just as important, as it can provide audio cues that provide the user with important information; the fire team member who hears the footsteps of the next member of his team right behind him does not need to turn around to know that the fire team member is right there, and this cue needs to be the same in the virtual world as it would be in reality.

- Voice over IP

In addition to sound from other objects in the environment, it can be important to provide voice audio cues, as well. In applications where team members are operating together, in order to provide a realistic, immersive environment, and to prevent the artificial situation of typing voice commands, Voice Over IP (VOIP) capability can also be important. Simulation engines can integrate VOIP so that it can be used, if needed, or left out completely, as it can be very resource intensive.

- Level editing tools

The process of placing objects and editing the states of all objects in a scene can be time consuming and overwhelming, especially for a large-scale application that involves many entities in the scene. Visual representation of where entities are, and the ability to make modifications, additions, and deletions to entities and to the scene can save the application programmer many tedious hours carefully manipulating the scene in order to put entities in the right place. Additionally, configurations can be created visually and saved as configurations for future applications, as well, making level editing tools not only time saving for current applications but for multiple applications.

- Graphical User Interface – Front-end

In addition to the graphical method of creating scenes, the addition of a graphical user interface from which to create front-end interfaces (i.e., introduction screens, main and auxiliary menus, end-of-level screens, help screens, etc) can also save time. In many instances, the application programmer does not even need to understand lower level windowing or Graphical User Interface (GUI) programming—this is handled through the GUI API in the engine.

- Ability to make user modifications (called Mods)

Once an application is built on a simulation engine, the ability to modify that application, as opposed to creating another application, can save time and resources in the development of a similar application. In many instance of training, the scenario may remain the same, while all that is needed is a different environment. Or the reverse could hold true, such that the same environment could be used for a different scenario, and the only changes needed are a different rule set for handling interactions in the application. In these and many other cases, the ability to create mods, or applications which are nothing more than modifications of similar applications, can be very useful and efficient.

- Scripting ability

In addition to providing the ability to make mods, some engines also provide scripting language support, so that a higher abstraction can be maintained and used by the application programmer in the creation of new application. This not only allows the application programmer an easier interface to the engine, but can also be used to ensure that the programmer using the scripting language does not create buggy code, by keeping the lower level code out of reach.

2. Commercial Off-the Shelf Software (COTS) vs. Open-Source software

In determining whether to use open-source software or to contract or buy commercial software, an important factor is the maturity of the particular open-source software versus the commercial off-the-shelf software (COTS). It makes sense to say that the DoD should use the best-value software products that meets its needs, whether those products are open-source or commercial. Open source software may be cheaper, but if it is not supported well enough for the DoD to effectively use the software and the DoD determines that supporting the software itself would be too costly, then it is not the best product.

a) Commoditization and Interoperability

However, the DoD is not an island; interoperability among services and with the commercial world is a key factor to the DoD's ability to do business. If software is not yet commoditized, one of the problems in choosing an open-source software solution is that the most interoperable solution is most likely the stable and well-established commercial product in that technology, not the open-source solution. Lack of interoperability costs money to overcome, avoid, or circumvent, and, thus, removes the benefits obtained through use of open-source software.

Eventually, however, most software technologies (web browser, office suite, 3d simulation engine tools, etc) reach a level of commoditization, at which point all products, both commercial and open-source, should provide the same basic functionality. Once a software technology has been commoditized, and interoperability is no longer a roadblock, mature open-source software can save the DoD money while allowing the flexibility to modify and maintain software as necessary. Certainly, in the specific application technology of simulation and game engines, commoditization has occurred. The user—the application programmer who will use the engine—can choose from a variety of options, some commercial and some open-source, and be reasonably assured that they will all provide the same basic services in the same basic manner. Applications created on one engine, will look much the same as, and will interact with, applications created on another. According to Dollner and Hinrichs, "Scene graphs are ubiquitous: most high-level graphics toolkits provide a scene graph application programming interface (API) to model 3D scenes and to program 3D applications. […] The generalized scene graph supports application development on top of different, independent rendering systems, integrates seamlessly rendering techniques that require multipass rendering, and facilitates declarative scene modeling." (Dollner and Hinrichs)

b) Cost Factor

Shifting to an open-source alternative to commercial 3d simulation engines can save the DoD money by removing expensive software license fees. It is true that hidden costs exist in terms of having personnel capable of maintaining open-source applications, but, in reality, the DoD currently receives contracted support for the commercial visual simulation software packages it uses; the same type of contracted support can be established for most well-developed and stable open-source solutions, while still not paying for software costs. Even with a contract for support, most open-source solutions would be less costly than the proprietary alternative.

Multigen-Paradigm Vega®[8] costs about seven thousand dollars for a single development license (on one computer), and about another thousand dollars for the runtime license needed for that same computer to be able to run Vega applications. That cost covers the basic license, but does not include the licenses—both developer and runtime—for any of the additional packages that provide functionality not found in the basic package, such as night vision or special effects software. In contrast to the expense of Vega, which costs a fee per computer developed and run on, other commercial engines, such as the Unreal® engine, charge a large enough fee up front for unlimited development and an annual service charge as well. In contrast to all of these fees, the source code for OpenSceneGraph can be downloaded for free, and a developer support userlist is available, as is paid development consulting by OpenSceneGraph foundation members.

c) Software Lock-in

In determining whether open-source software breaks the lock-in mode established by buying commercial software, the question must be asked, "Can we replace this choice with another and still have the same functionality and usability?" Because open-source choices can answer yes to this question, lock-in is removed by moving to OpenSceneGraph, or any other open-source alternative. Even in the absence of commercial alternatives, alternative software can be developed to replace existing software. Additional functionality needed can be added to the open-source solution; in fact, the flexibility exists to either add functionality using organizational man-hours or to contract programming services, still at lower cost than to purchase commercial alternatives. An added benefit to this open-source architecture-type solution is that if there are better-value commercial products that can replace part of the open-source solution, (in the case of OpenSceneGraph, a best-value commercial physics engine), the best-value pieces can be integrated into the open-source architecture.

While open-source does remove the lock-in barrier, it needs to be considered carefully. There are legacy systems in DoD software for which code is available, but for which the man-hours in maintaining or upgrading the software is

[8] Vega® is a registered trademark of Multigen-Paradigm; http://www.multigen.com/.

prohibitive. There are hidden dangers of backing into a corner from which the solution is ultimately more costly than having stayed with a commercial mainstream product. This goes to the issue of maintaining the best-value solution. Certainly, because open-source removes the lock-in barrier and poses a good-value alternative, it forces commercial vendors to make a better product.

F. SPECIFICATION OF SOFTWARE ENGINE FUNCTIONALITY

1. Superset of Software Engine Modules

Modules that are desirable in current commercial game and virtual environment simulation engines grow at a rapid rate. New technology is constantly driving the addition of new features into the engines of today. Many of those features—in addition to the basic scene graph capabilities—are included in the following list:

- Effects system
- Physics engine
- Character animation
- Game networking
- Scripting ability
- Artificial Intelligence for agents
- Sound, including spatial/positional sound
- Voice over IP
- Level editing tools
- Graphical User Interface – Front-end
- Ability to make user modifications (called Mods)

2. Modules Covered in this Thesis

Chapter II of this work discusses the modules that the authors added to the libGF architecture. These are not the only additions made by the authors, but are the largest and most significant in impact, and need to be discussed in detail so as to provide not only an understanding of the motivation behind adding these modules to the libGF architecture, but also the method by which they were implemented as well as their application. This provides the reader with, first, an understanding of how to create similar modules, how to add similar modules into similar architectures, and how to change the module itself within the libGF architecture, and, second, how to use the described modules in new applications. The reader is reminded that a libGF Quick-Start Users Manual can be found in Appendix A.

The modules discussed in Chapter II include:

- Character animation
- XML read/write capability
- Agents
- Networking
- Physics

II. INPUT IMPLEMENTATION

A. CHARACTER ANIMATION

1. Motivation

a) Introduction - What is Character Animation

Character animation is defined as applying motion to an inanimate or modeled character to express emotion and/or behavior. This can be done on anything from something as simple as paper and pencil to state of the art computers hardware. Today, computers are used to a great extent to increase the realism, speed and range of animations. Character animation has found its way into everything from games to virtual environments to movies. But, it wasn't always that way. In investigating the process of character animation, it’s important to take a brief look at the history of animation.

b) History of Character Animation (Pre-Computers to Present)

Man has strived to capture and express motion in all types of art forms since the beginning of time. The early cave man tried to represent the animation of the hunt for food on the walls of their dwellings. The Egyptians drew extensive pictures of the pharaohs in pyramids trying to express different forms of motion, one example being the drawings of men in different wrestling stances on panels found from circa 2000 B.C.

Inventions in the early 1800's showed man's understanding of the principle of animation called persistence of vision. Paul Roget's invention, the thaumatrope, demonstrated how images are retained for a duration of time by twirling a disc with an image of a bird on one side and an empty cage on the other. When spun, the two images were combined to show an image of a bird in the cage - a good example of persistence of vision. Additional inventions by Joseph Plateau and Pierre Desvignes helped to increase man's understanding of animation and helped pave the way for what animation has become today.

Starting in the 20th century, significant efforts started to emerge in the field of character animation. In the early 1910's, Winsor McKay started his work on single person animated films and in 1914 he finished work on "Gertie the Dinosaur." This film along with his other work in the field of animation earned him the title of "father of the animated cartoon."

About the same timeframe, animation studios started cropping up and work in animation strayed away from being done by a single person and towards the "streamlined, assembly-line process in the best Henry Ford tradition." (Crandon, 1999) Early animation work that appeared from around the 1920's on, included:

- Felix the Cat - created by Otto Messmer in 1919
- Mickey Mouse - Walt Disney Studio in 1928
- Looney Tunes - Warner Bros. in 1930
- Daffy Duck - Warner Bros. in 1937
- Porky the Pig - Warner Bros. in 1938
- Snow White and the Seven Dwarfs - Walt Disney Studio in 1937
- Pinocchio - Walt Disney Studio in 1940

A lot of these animations were accomplished by drawing individual cels and were plagued with somewhat unrealistic looking character movements. Because of this, Disney went towards tracing animations over film footage to obtain more realistic animation for its film Snow White. In the 1970's, as computers became more capable, animations started to be accomplished on the computer vice by hand.

Starting in about the 1980's, the use of motion capture was looked at for creating more realistic character motion. Notable work in the early work with motion capture included:

- Tom Calvert used potentiometers to drive computer characters (early 1980's)
- Optical tracking systems appeared - Op-Eye and SelSpot (early 1980's)
- Graphical Marionette showed the "scripting-by-enactment" technique - MIT (1983)

- Mike the Talking Head demonstrated the ability to control facial movements - Silicon Graphics (1988)
- Waldo C. Graphic: a face controlled by an 8D mechanical arm (1988)
- PDI developed an "exoskeleton" system for tracking upper body movements
- Dozo created by Kleiser-Walczak using optical motion capture system (1989)
- Face Waldo, which allowed real-time performances of Mario (1992)
- Acclaim demonstrated two person animation completely done by motion capture (1993 SIGGRAPH)

Since the start of animation being generated via computer, the amount of work being animated has continued to grow by leaps and bounds and the quality has become extraordinary. It is approaching the time when computer generated characters will no longer be distinguishable from real characters.

c) The Principles of Animating a Character

Before looking at the process of creating a model, or the methods of applying motion, it is necessary to be familiar with some principles that go into creating character animation. The 12 principles that follow are summarized from Michael Comet's "Character Animation: Principles and Practice." (Comet, 1999)

- Timing - Timing is everything in animation. Changing the timing of one movement can change the emotion or behavior expressed.
- Ease In and Out - Rapid movements are not always what is desired. Starting to walk or coming to a stop should involve a gradual ease into or out of instead of a rapid walk starting from a complete stop.
- Arcs - Almost all movement in the real world involves arcs, vice straight lines - so should animations.
- Anticipation - This means that animations should allow the viewer to know what is going to happen next.
- Exaggeration - Used to accent an action to appear more realistic.

- Squash and Stretch - Objects in motion are affected by squash and stretch, so should the objects in animations. Balls squash down when hitting the ground and muscles bulge when used.
- Secondary Action - A good example for secondary animation would be a horse running and coming to a quick halt before reaching the ledge of a cliff. As the horse comes to a quick stop, the ears and tail should be affected by the stop of forward motion, this is the secondary action.
- Follow Through and Overlapping Action - This is the action at the end of the action. When throwing a ball, the arm doesn't stop when the ball is released, but continues on as follow through. The same is true for most other actions.
- Straight Ahead Action and Pose-to-Pose Action - The difference is this: straight ahead action is drawing each frame from the start of the motion to the end of the motion, while pose-to-pose action is defining the start pose and the end pose and allowing the in-between poses to be interpolated.
- Staging - Make sure that what the viewer is supposed to see is easily understood. Don't have too many things going on at the same time.
- Appeal - The shapes and motion should be appealing to the viewer.
- Personality - By combining all the above principles and the animations themselves, the character will take on a life of its own, and with that its own personality.

Following all these principles doesn't guarantee a good character animation, but it is instead a good starting point. The creation of character animation still involves having patience, skill, a good model and a good technique for applying motion to the model. The first step in that process is creating a model.

d) Modeling the Character

There are several different ways to create a 3D model, which one you choose depends on your skills and what the desired use is. Let's look at a few ways that models are created:

- By Hand - Polygon by Polygon

This technique can be very time consuming, but can be a good starting point for just starting out in 3D modeling. The process is to start off with a reference point, like a photograph, and then start plotting points where they should be in reference to the photo. The points are connected with polygons forming a mesh in the shape of the reference photo. When the mesh is complete, a texture (usually from the photo) is applied the polygons, hopefully giving a pretty accurate model of the original photos.

- 3D Scanning.

The approach this technique uses is high-resolution photography. A large number of high-resolution photos are taken of the object wanting to be modeled from every angle. The photos are then stitched together via software to produce a 3D model. The high-resolution photos also produce the textures necessary for the model, creating a photo-realistic model. This approach is usually done through 3D scanning studios and incurs a cost.

- Digitizing

This appears to be a favorite among animators for creating 3D models. The process starts with making an actual model of the character, usually in clay. Lines are drawn on the clay model forming a kind of mesh over the model. Using a digitizing arm, the lines are then traced, which creates the 3D model in the computer.

e) Creating the Animations and Motion Capture

Animation is created using a variety of different methods, each having its advantages and disadvantages. This section looks at a few of the techniques that animators use to create animations for use in computer generated (CG) characters. All animations, though, usually stem from the observations of real movements, whether it is from humans, or animals in nature. What better way to model how a human walks than to observe how humans actually walk, or to find some way to capture that walk, or the "data" of that walk. When modeling horses or birds, to be able to create realistic looking animations, research and observation are required. Industrial Light & Magic animator

Dan Taylor explains the research that went into the animating of the dinosaurs in Jurassic Park, "Although there were lengthy discussions with paleontologists about the probable appearance of dinosaur walk and run cycles, animators looked to bird and reptilian movement for inspiration." (Street, 2002)

However, simply observing movements is not enough. The process of video taping a person falling and then playing back that person falling is not considered motion capture. In order to have motion capture, the "motion" of the person falling (or more specifically, the data associated with the falling) has to be captured. As John Dykstra explains the process for the making of Spider-Man, "We studied Tobey's posture, movements, and how he gestured...and we translated all those details into the actions and maneuvers of our virtual Spider-Man." (Scott, 2002)

The major categories found today for creating motion can be divided into: manual specification or keyframing, procedural and simulation, and motion capture. All three are currently used in the process of creating motion by animators.

- Manual Specification or Keyframing.

This method entails creating a series of individual poses and then creating the animation via a series of keyframes of those poses. "Keyframing has been the traditional approach to controlling the subtle details of virtual humans. However, the technique requires the animator to have a detailed understanding of how moving objects should behave over time, in addition to the skills needed to produce the key frames." (Millar, 1999) This can be a long and tedious task. The skill level required for this technique is high and usually requires patience and training. This method sometimes proves to be a daunting task to create realistic looking animations. Animation software has gotten better and helps to alleviate some of the task by performing interpolations in between keyframes. Some of the features that have been introduced in the past few years include:

  - Patch-based animation allows smooth, flexible movement.
  - Complex movements are simplified; unique bones motion offers lifelike bouncing and twisting.

- Complete skeletal and muscle control features.
- Inverse Kinematics (IK) for creating skeletal based motion.
- Character animation with lip-synch made easier.
- Stride length to prevent a character's feet or tires from slipping as they move.
- Action Overloading; applying layers of Actions to a character so that it can "walk", "talk", and "clench" its fists simultaneously.
- Action Range; choose only a range of frames, Hold, or Wait from an Action
- Rotoscope facial movement in Muscle with sequenced backgrounds.
- Poses.
- Lock Bones.
- Sophisticated key frame controls.

Even though the new features greatly aid the animator in creating the keyframe animations, the process is still slow and sometimes frustrating. Tomek Baginski recounts his experience in animating his animated short The Cathedral, winner of Best Animated Short at SIGGRAPH 2002's Electronic Theater, "My first walk cycle took me three weeks and it still looked bad. But when I finished, months later, I was able to animate one scene in a day and it looked better than [motion capture] " (Animation Magazine, Jul 2002). The animated short took Tomek 14 months to complete and was animated using 3DS Max.

- Motion Capture

Motion capture provides a very accurate method for capturing character motion. It is being proven extremely valuable in a variety of applications, some of which are being utilized today are:

- Gait Analysis

- Physical Rehabilitation
- Sports Medicine
- Sports Analysis and Performance Enhancement
- Entertainment - Live Performance and Pre-Visualization
- Prototyping
- Character Animation (used in real-time simulations, movies, games, television, videos, industrial training, etc)

The use that we are obviously going to concentrate on is for character animation. The most common techniques that are being used for motion capture are magnetic tracking and optical tracking.

Magnetic systems used for motion tracking utilize sensors that sense a magnetic field inside a set volume and provide a means of accurately determining the position and orientation of the sensor. Earliest implementations had to deal with several performance degradations, specifically:

- used cables attached to the sensors, limiting the free movement and range of the user in the capture volume
- were susceptible to sensor noise and drift
- had to overcome electrical and metallic interference from nearby objects
- were difficult to use
- problems getting accurate data when actors became too close together

Current advances have brought improvements to the magnetic sensor systems and have provided for:

- wireless connectivity, increasing the range and allowable free movement of the person

- increased performance
- minimized interference and distortion by switching from AC to DC
- easier and faster setup process
- providing 6 degrees of freedom
- allowing for multiple characters being captured simultaneously

The sensors for the magnetic systems are easily placed either inside a custom body suit or can usually be fixed near joints on the outside of close fitting clothes. With the advent of the wireless capability, a small computer can be worn on the back of an actor, the sensors attached to the computer and the computer transmitting via 802.11b to a capture station. The data can be captured for cleaning and later use, or can be applied to real-time digital characters.

Optical tracking is broken into passive and active sensors. Active sensors are implemented by light emitting diodes (LEDs) and passive sensors utilize retroreflective spheres. Both are common in today's motion capture industry, but passive seems to be the more dominate in use.

Passive sensors work by placing small retroreflective spheres on different parts of the body and then detecting the movement of those spheres by detecting the light reflected from the spheres. The more the markers and the better, or higher resolution, the cameras are, the better the capture is going to be. Problems associated with passive optical devices are the need for high-resolution cameras, the occlusion of spheres due to objects and body parts in the capture volume and the lack of being able to identify individual markers. Propriety software is often used in the post-capture process to identify markers between frames.

Active sensors utilize LEDs as the target markers, which allow each marker to be individually identified. This means accurate tracking at usually higher frame rates, but it does require larger trackers vice camera to capture the data.

An additional method of providing motion capture, besides the magnetic and optical systems, is an electro-mechanical system that is worn like a skeletal suit and tracks the movements of the joints. Potentiometers measure the changes in voltage at each of the joints and provide a rotation value for that joint. This rotation provides the data necessary to model the motion of the limbs that the skeletal system is attached to. An example of this type of system is the Gypsy System by Meta Motion.

2. Implementation

a) Integrating the Character Animation API

Keeping with the premise of building an open-source and low cost architecture, the number of viable character animation libraries to integrate was greatly reduced. The library that was chosen and integrated in libGF is called Cal3D[9]. The use of Cal3D in libGF requires the inclusion of cal3d.h and the statically linked library, cal3d.lib. The wrapper to Cal3D are in the following three classes: gfCal3dObject, gfCal3dModel, and gfCal3dNode. The hierarchy is represented by gfCal3dObject containing a gfCal3dNode, which in turn contains a gfCal3dModel. When a new gfCal3dObject is created, by instantiating it as a new object and subsequently adding it to the environment, a new gfCal3dNode is created (shown in Figure 6).

```
humanNode = new gfCal3dNode("gfCal3dNode", mFileName);
```

Figure 7. Creating a new gfCal3dNode

On instantiating the gfCal3Node, arrays are created to handle the processing of the geometry arrays based on the number of geometries in the actual character model (shown in Figure 8).

```
geoCoordinateArray = new gzArray<gzVec3>[m_numGeometries];
geoNormalArray = new gzArray<gzVec3>[m_numGeometries];
geoPrimitiveLengthArray = new gzArray<gzULong>[m_numGeometries];
geoIndexArray = new gzArray<gzULong>[m_numGeometries];
geoTexCoordinateArray = new gzArray<gzVec2Vector>[m_numGeometries];

//add all of the geometries to the model (once)
addGeometries();
```

Figure 8. Creating the geometry arrays in gfCal3dNode

b) Modeling the Character

[9] Cal3D is an open-source character animation library found at http://sourceforge.net/projects/cal3d

Characters used inside a virtual environment have to be modeled before they are inserted into the environment. The process that is used to generate the character is different for each of the modeling applications that exist, but the general concepts remain the same. Whether the character is modeled using polygons or Nonuniform Rational B-Splines (NURBS), the end result is that the model needs to be in a polygon form before the character can be used in the environment. Since the VEs in use today—and the ones built using libGF—can be very graphics intensive, and the animations of the character are seen in real-time, the polygon count of the modeled character should remain low. High polygon count models are useful when the desired result is photo-realistic models, but in a virtual environment, where performance of real-time animations is an issue, polygon count matters. The typical polygon count used in games and virtual environments is around 5,000 polygons per model. The model that was used for character animation in this thesis has a polygon count of less than 2,000 polygons.

This thesis is not geared toward covering the techniques used for polygon or NURBS modeling, because there are numerous books written on the subject and it is very dependent on the 3D modeling application used. The application used is cost dependent; the sophisticated, commercial 3D modeling packages cost thousands of dollars, while the cost of alternatives range from low-cost to free. The packages that were used for this thesis were Discreet® 3ds max™[10] and Alias® Maya®[11]. The same model could be created in a free or low-cost 3D modeling application such as Blender[12], Wings3D[13], or MilkShape 3D[14].

Regardless of the package chosen to create the character, in order to be able to apply animations, the character should be modeled in a standard pose, such as the

---

10 3ds max™ is a registered trademark of Discreet®. Discreet® is a subsidiary of Autodesk, Inc. – http://www.discreet.com

11 Maya® is registered to Alias Systems, a division of Silicon Graphics Limited - http://www.alias.com

12 Blender is a recently turned open-source 3D modeling application from the Blender Foundation – www.blender3d.org

13 Wings3D is a free polygon mesh modeler - http://www.wings3d.com

14 MilkShape 3D is a shareware application by Chumbalum Soft - http://www.swissquake.ch/chumbalum-soft/index.html

T-pose shown in Figure 9. Careful attention needs to be paid to ensure that the model is complete, with no gaps between vertices, and that duplicate vertices do not exist. Both conditions will result in unfavorable mesh transformations and appearance of the model when the animations are later applied.

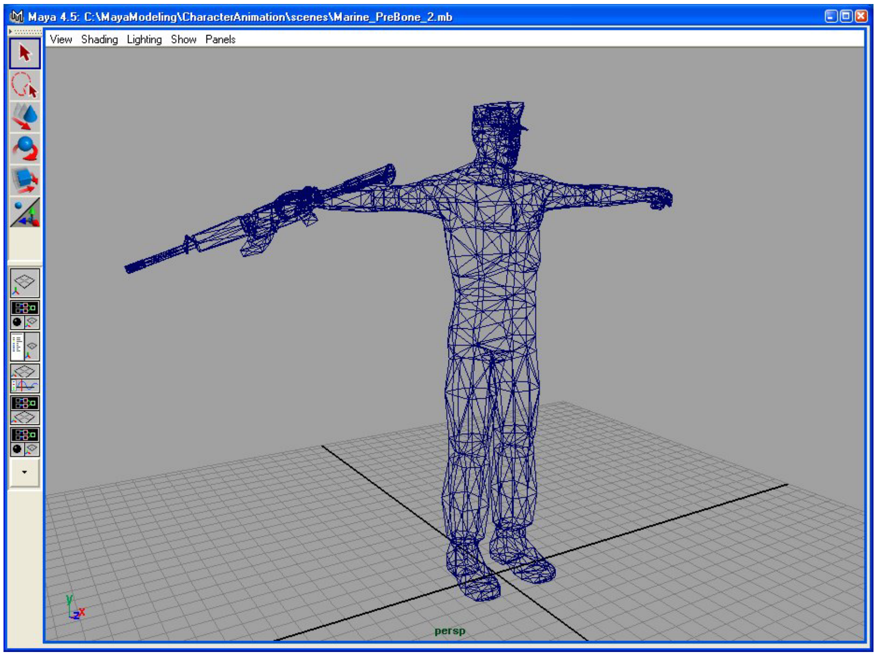


Figure 9. Modeling the character - polygon mesh

Once the polygon mesh is complete and the character is correctly modeled, a texture is applied to give the model the desired appearance. Since this thesis deals specifically with military applications for virtual environments, the texture chosen was one of a Marine. The normal practice for applying textures to a model is to have all necessary textures combined into one texture file from which the different parts of the texture are applied—using UV coordinate mapping—to the appropriate portions of the model. Figure 10 shows what a combined texture file looks like, with the final result of the texture being applied to the model shown in Figure 11.

Figure 10. Texture file before applying to model

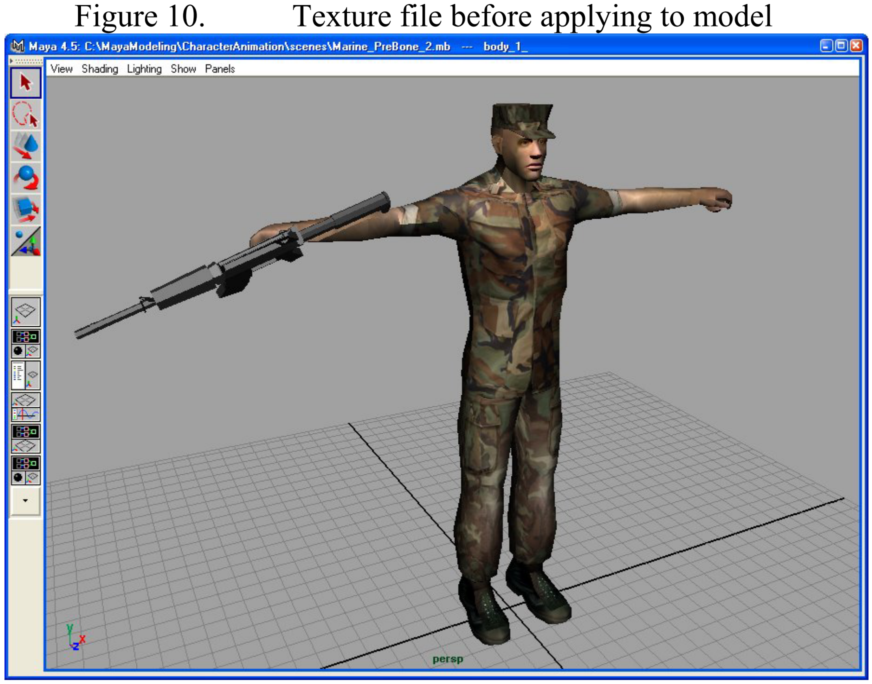


Figure 11. Texturing the model

Since the model is going to be used in a virtual environment, and because most applications assume the orientation, it is best to have the character model facing in the +Z direction.

c) Rigging with a Skeleton

Rigging a model is the process of building a hierarchy of bones to build a skeleton for that character and then applying the polygon mesh to the skeleton—called skinning. Most animators start with a somewhat standard bone hierarchy and then modified the setup to suit their needs. If the character animation is going to include hand animations, then the bones in the hand are important; however, if there will not be any fine-tuned hand movements in the animations, then there is no need to insert the extra bones. The skeleton should be simplified to what is necessary and functional for the animations will going to be generated. Additional bones are often used for points to add equipment, apply facial animations, and attach weapons. These bones provide an easy insertion point and a means to provide new animations.

The character model used for this thesis required only a standard human skeleton rig and an additional bone to facilitate the use of a weapon. This additional bone joint allowed the weapon to be easily hand animated—key framed—separate from the animations applied to the entire skeleton. By defining constraints, single joints can be made to follow the same movement, and/or orientation, of a single point or another joint. Constraint addition, with respect to the weapon, makes it easy to force the weapon joint to stay constrained to the right shoulder joint, and at the same time always orient itself to a point in front of the character.

An additional consideration when creating a character skeleton is whether it is necessary to provide the additional joints for possible limb rotation. For example, when creating the right arm hierarchy, one way is to create the right shoulder, right arm, right forearm, and right hand. This doesn't necessarily lend itself to easily isolate and rotate the right forearm, except from the joint between the right arm and right forearm. The solution, if that kind of separate rotation is necessary, is to provide additional joints between the right arm and the right forearm; an additional bone called 'right forearm upper' would allow the right forearm and right hand to be rotated separate from the right arm. Although this functionality was not necessarily required for this thesis work, the additional joints were added for possible future functionality.

Figure 12 shows the hierarchy that was used in the rigging of the model. The hip joint acts as the central joint for the entire skeleton, with the upper body (spine, head, and arms) and both legs branching off. The final rigged character is shown in Figure 13. The model is in the standard T-pose and the weapon has been moved out from of the character and attached to the weapon joint.

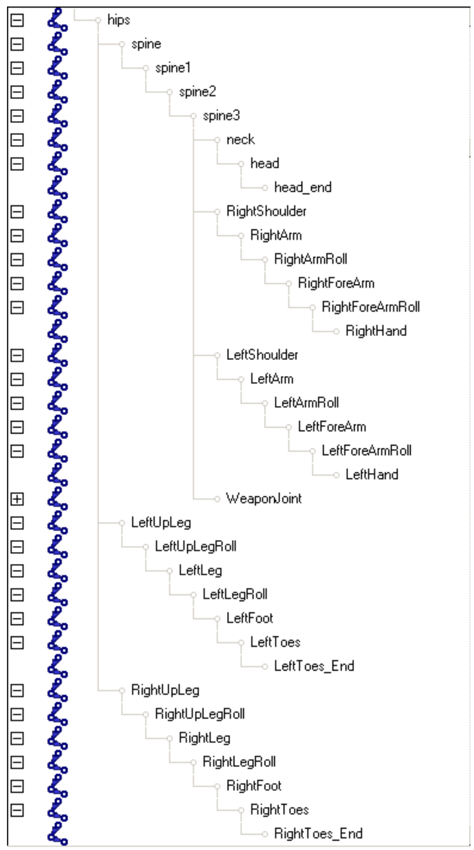


Figure 12. Hierarchy of bones in character model

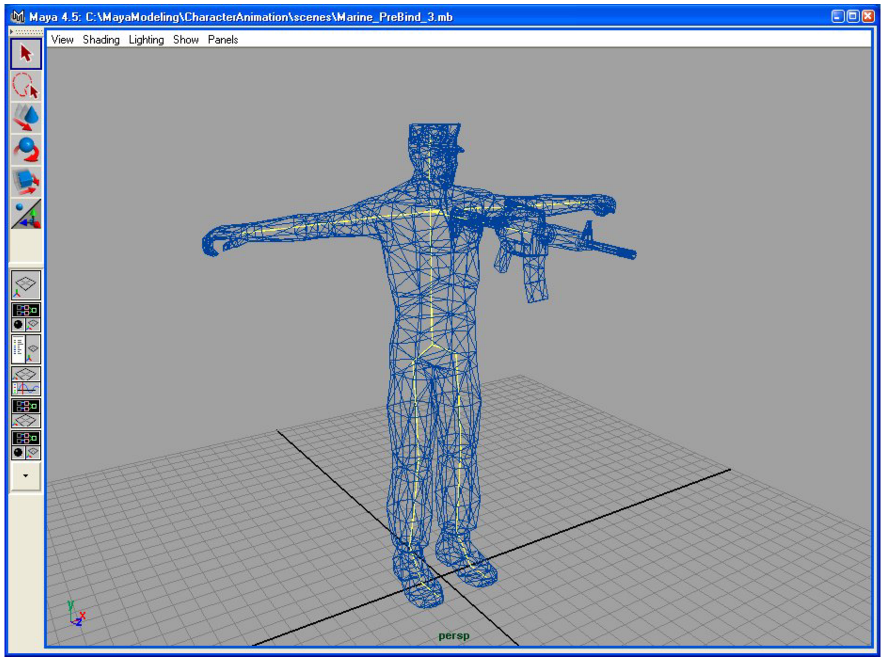


Figure 13. Rigging the skeleton

Prior to applying weights to the vertices, the mesh must be bound to the skeleton. This process is called skinning the model and should be done using a smooth bind. Usual settings when applying the smooth bind include choosing the complete mesh to bind and for binding mesh vertices to the closest joint.

d) Applying Weights to the Vertices

In order for the mesh to act appropriately when the skeleton is transformed and animations applied, proper vertex weighting has to be applied to the mesh. This process tells each vertex how much influence is contributed from the movement of each of the joints in the skeleton. If a contribution from a foot joint is attributed to the mesh surrounding one of the shoulders, then when the foot joint is moved, the mesh around the shoulder will show deformations. This is obviously an unnatural behavior and is fixed by proper weighting of the vertices.

The vertex weights are modified by editing the smooth skin bound to the skeleton; Figure 14 shows the character model with the LeftArmRoll joint selected and the vertex weights which that joint contributes to. The more joint contributions placed on a vertex, the whiter the mesh appears around that vertex—while in vertex paint mode. Notice that the rest of the body, besides the vertices in the vicinity of the upper left arm, are shown in black—meaning that the LeftArmRoll joint has no contribution to other vertices.

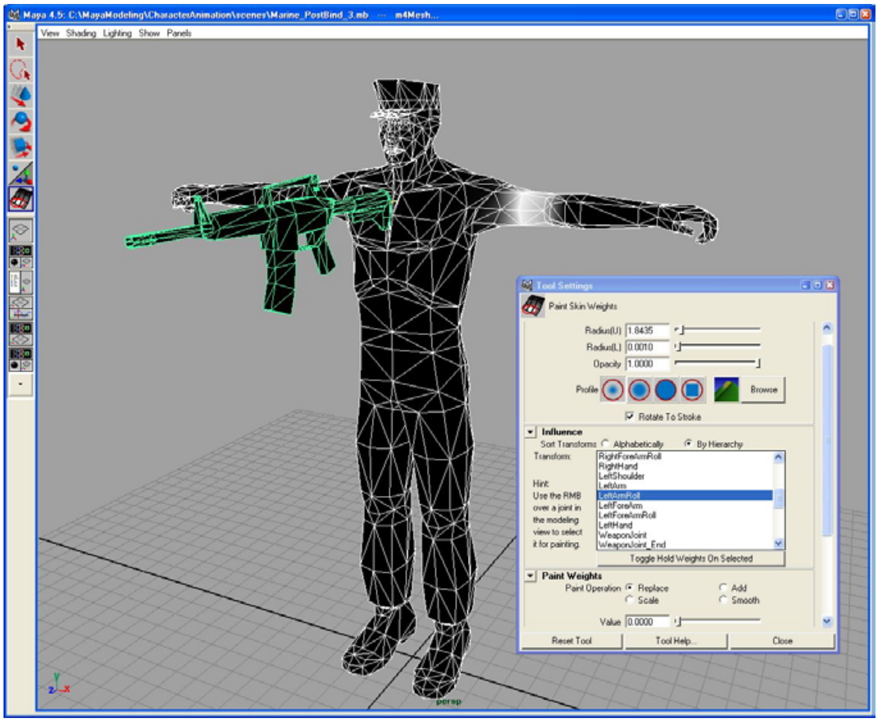


Figure 14. Painting the vertex weights

While applying vertex weights to the mesh, prior to animating the model, the joints should be moved and rotated to verify that the correct weights have been applied to the vertices affected by each joint. Doing this allows missed and incorrectly weighted vertices to be fixed during the weighting process. Once the character model has been rigged and the vertex weights applied, the model is ready to animate.

e) Mapping and Applying the Motion Capture Data

In order to apply animations to the character and avoid keyframing all animations by hand, Motion Builder™[15] can be used to easily apply motion capture (mocap) data to the skeleton. Exporting the character model from a 3D modeling application to FBX[16] format—the industry standard—allows the model to be imported into Motion Builder™, which can apply mocap data.

---

[15] Motion Builder™ is a registered trademark of Kaydara, Inc. – www.kaydara.com

[16] FBX is a cross-platform file interchange format for 3D data, widely used in the animation industry

The motion capture data is mapped to a character using the steps outlined below. This is by no means an exhaustive list of the steps necessary, but more of a synopsis. A complete procedure is found in the Motion Builder™ and readily available tutorials found at 3D Buzz.

- Inside Motion Builder™, first load the character model from the FBX file used to export it from the 3D modeling application.
- Make the model a character by dragging the Motion Builder™ character icon located in the asset browser onto the model. This will create a new character and, provided the skeleton was laid out correctly, will characterize the character allowing it to be used inside Motion Builder™.
- Create a control rig for the character.
- Ensure that the foot definition markers are aligned properly and placed so that they contact the floor.
- Drag an actor from the Asset Browser onto the character.
- Import the optical motion capture data and position the virtual actor so that it aligns with the optical markers.
- Create a marker set, which attaches the optical markers to the Actor.
- In the character control tab, set the input to be from the Actor.
- Now playing the motion capture data, attached to the Actor, will also animate the Character.
- Modifications to the animations can be made using the bend and rotation settings of the character.
- Once the animation is complete, it must be backed onto the character.
- The Character is now ready to export back out to an FBX file and imported back into the 3D modeling application for fine-tuning.

Figure 15. Mapping the motion capture

When the mocap data has been applied and the desired animations are ready, the model is once again exported to the FBX file format and then is re-imported into the 3D modeling application. There, the animations are fine tuned and cleaned to produce the file animations that will be exported for use in the virtual environment.

f) Final Adjustments and Exporting the Model

Once the model is back in the 3D modeling application, the animations are fine-tuned, with possibly new ones added. In the case of the Marine model, the animations that were applied to the weapon inside Motion Builder™ did not prove to be very realistic; so, the keyframes specific to the weapon were deleted and the weapon joint was keyframed by hand. This allowed applying the constraints discussed above in Section b to always orient the weapon with the right shoulder and always aim at a locator placed in front of the model. Both hands were also positioned to appear to be always gripping the weapon, even with the normal bouncing and swaying that occurred during the walk and run animation cycles. Making these fine-tune adjustments resulted in a more realistic character animation.

Since Cal3D was the animation package integrated into libGF, the model and its respective animations had to be exported to a format compatible with Cal3D. Currently, there are exporters available for 3ds max™ and Blender, with an exporter for Maya® in progress. To export the model and animations from 3ds max™ using the Cal3D exporter, the following steps must be taken:

- Load the model into 3ds max™. Make sure the animations are cleaned up and fine-tuned.
- Ensure that the Cal3D plug-in is installed.
- Make sure the character is in the base pose (T-pose).
- Select the skeleton and export using the Cal3D exported, selecting the Cal3D Skeleton File (*.CSF) option. Make sure all the bones are selected prior to exporting.
- Select the mesh (with the character still in the base pose) and export again, but this time select the 'Cal3D Mesh File (*.CMF)' format. You will need to select the previously exported skeleton file to associate with prior to exporting.
- After making sure that the animations are good, export the animations using the 'Cal3D Animation File (*.CAF)' export option. Again, you will have to select the previously exported skeleton and ensure that all the bones are selected.
- Lastly, export the textures/materials by using the 'Cal3D Material File (*.CRF)' export option. You will need to select the appropriate material to associate with the model when exporting.
- Using the file format shown in section II.B.3.c, use the filename of the newly exported skeleton, mesh, animations, and material to prepare the new character for use in the virtual environment.

3. Application

Using character animation in an application written with libGF requires creating a new gfCal3dObject, as shown in Figure 16, and then attaching a motion model to the player as discussed in section F.3.a.

```
// Get the full pathname of the model XML file
char configFile[256];
gfGetFullFileName(configFileStr, configFile);

// Create the new gfCal3dObject
gfCal3dObject *avatarObject = new gfCal3dObject(objectNameStr, configFile);

// Set the scale of the character
avatarObject->SetScale(obj.scaleX, obj.scaleY, obj.scaleZ);

// Create a new gfPosition and use it to set the object's position
gzRefPointer<gfPosition> pos = new gfPosition(positionX, positionY, positionZ,
                                        positionH, positionP, positionR);
avatarObject->Position(pos);

// Add the new avatar object to the environment
environmentPtr->AddObject(objectNameStr);

// Find the motion model for the object, set the offset of the avatar and rifle, then set
// the avatar to receive messages from the motion model (for updating position and
// animations).
gzRefPointer<gfMotionHuman> motion = (gfMotionHuman *)gfFindMotion(motionStr);

if(motion != NULL) {
	gzRefPointer<gfPosition> offset = new gfPosition();
	avatarObject->GetOffset(offset);
	motion->SetRifleOffset(offset);
	avatarObject->AddNotifier(motion);

	// Make body and geometry for the motion model
	char marineCollisionFile[256];
	gfGetFullFileName("Marine_collision.xml", marineCollisionFile);
	gfCDGeomGroup *motGeomGroup = new gfCDGeomGroup("motGeomGroup",
		marineCollisionFile);
	gfPhysicsBody *motBody = new gfPhysicsBody("motBody");
	motBody->setMassToBox(10.f, 1.3f, 1.3f, 1.3f);

	// Set the physics pieces for the motion model
	motion->setCDGeom(motGeomGroup);
	motion->setPhysicsBody(motBody);
	motion->setSlip(0.1f);
}
```

Figure 16. Creating a new gfCal3dObject and adding the motion model

Once the new avatar object has been created, the interaction between the avatar and the user is handled automatically by the human motion model trapping the input from interface device and sending the necessary animation commands to the gfCal3dObject class; also over the network if that functionality is being used. Modification of the model definition file allows easily modifying the settings that are used to define, load, and configure the avatar model. The layout of the file is discussed in more detail in section B.3.c.

Once the animations are created, they can be applied to multiple character models. Following the steps discussed earlier, a new model can be created, textured, rigged, and animated and then exported for use in the environment model.

4. Future Work

The future work needed for the character animation library is the addition of dynamic blending of animations. Currently, the API does support blending but only static blending. A Marine walking animation and an animation of the weapon being aimed at 45 degrees up can be blended together to form the Marine walking and aiming the rifle. But there is no current method to dynamically blend the two together to make the Marine walking and aiming the rifle at 45 degrees down. In order to have the rifle aiming to be able to occur at all angles, each position of the rifle would have to be animated. Providing dynamic blending would help to increase the realism of the remote player's rifle aiming and would decrease the number of animations required to achieve that realism.

In addition, increasing the collection of animations is required to provide more realistic training such as crouching, jumping, etc. These animations exist in motion capture form but, because of the lack of a Cal3D exporter for Maya and the problems seen when trying to export the animations and model back into 3ds max for exporting, the animations were never added. With a working exporter for Maya these animations could be added to the library in a very short time. There is currently work being done on a Cal3D exporter for Maya and when complete it will help to reduce the workflow of model to animating to application.

B. XML READ/WRITE FUNCTIONALITY

1. Motivation

Data files have notoriously been custom, or proprietary, formats lending themselves to becoming unmanageable or even obsolete. As applications are written, the necessary file format is generated and often revised over the course of the application's lifespan. The files are usually stored in either a cryptic text format, or a unknown binary format. Both formats prevent easy modification by a user outside the application and often lead to application errors due to of incorrect, or unknown, data.

In 1996, the W3C introduced a new file format, built upon and extending the Standard Generalized Markup Language (SGML) file format, called Extensible Markup Language (XML). This was created as a meta-language and imposes very strict formatting requirements on the file formatting. The data of an XML file is easily understood because descriptive tags are stored with the data, i.e. data describing the data. XML files are composed of a hierarchical structure comprised of text fields, known as entities. This strictly formatted hierarchy of human-readable data makes it easy for data to be entered, modified, and interchanged.

Our decision to implement XML as the data file format was because of the ease of use, the readability and known format, ease of modification by a generic text editor, and the relative ease of parsing the files.

2. Implementation

There are several open-source XML parsers available for use. We looked at a couple and decided that only a simple, sequential read/write parser was required. The XML library that we chose to integrate was ParamIO written by Arnaud Brejeon. The only limitations that we ran into when using the library was the fact that entities can not share the same name—meaning that you can have a structure such as:

```
<MediaPaths>
    <Path>\data\textures</Path>
    <Path>\data\images</Path>
</MediaPaths>
```

Figure 17. Incorrectly labeled entities

This limitation is easily overcome though by simply modifying the entity tags to contain a sequential number for each new entity, as below:

```
<MediaPaths>
    <Path00>\data\textures</Path00>
    <Path01>\data\images</Path01>
</MediaPaths>
```

Figure 18. Correctly labeled entities

The integration of the ParamIO library into libGF was done with only slight modifications made to the library. Instead of creating a new libGF wrapper class around ParamIO, the methods are set to be directly called; making the integration virtually transparent.

3. Application

As stated previously, we chose to use XML as our standard data file format. This allowed us the flexibility of being able to modify most aspects of the application specific data, such as media paths, the use of trackers, and even the models used, without having to modify the application code and perform a re-compile. A simple change to an XML file and rerunning the application is all that is necessary to completely change most aspects of the scenario being run. The ease of opening an XML file and reading in the data is shown in the following snippet:

```
ParamIO inXml;

inXml.readFile(filename);  // Read the file from disk

// Read in media paths
inXml.read("System:MediaPaths:Number", m_numMediaPaths, 0);

char mediaPathStr[64];

for(int x = 0; x < m_numMediaPaths; x++) {
   sprintf(mediaPathStr, "System:MediaPaths:Path%.2d", x);
   inXml.read(mediaPathStr, m_mediaPaths[x], std::string(""));
   printf("Loading in media path... %s\n", m_mediaPaths[x].c_str());
}
```

Figure 19. Example XML code

The data files that we chose to implement are as follows (with accompanying examples):

a) *System File*

The system file (default name is *system.xml*) is where the majority of the application specific configuration is done. All the normal settings dealing with application specific attributes are outlined in the example listing below. The key settings that are addressed in this file are network configuration, media paths, player and observer settings, window and channel settings, motions model used, and the configuration of inertial trackers (if used). A key point about configuration using XML, is that if a configuration is being used that does not require the use of trackers, but they are being used in a different situation, instead of removing the entire section from the file, simply change the number of trackers field to ‘0’ and no tracker information will be read-in by the application.

```
<System>
        <Name>CQBTrainer</Name>
        <Network>
                <Mode>standalone</Mode>
                <Server>ripple</Server>
                <Player>GoodGuy</Player>
                <Notifier>
                        <Type>gfMotionHuman</Type>
                        <Name>Motion</Name>
                </Notifier>
        </Network>
        <MediaPaths>
                <Number>3</Number>
                <Path00>c:\\Models\\</Path00>
                <Path01>c:\\libgf\\data\\Marine\\</Path01>
                <Path02>c:\\libgf\\data\\Models\\</Path02>
   </MediaPaths>
        <Windows>
                <Number>1</Number>
                <Window00>
                        <Name>window1</Name>
                        <Size>
                                <Left>50</Left>
                                <Right>850</Right>
                                <Top>50</Top>
                                <Bottom>650</Bottom>
                        </Size>
                </Window00>
        </Windows>
        <Channels>
                <Number>1</Number>
                <Channel00>
                        <Name>mainChannel1</Name>
                        <Window>window1</Window>
```

```
				&lt;Color&gt;
					&lt;R&gt;0.5f&lt;/R&gt;
					&lt;G&gt;0.5f&lt;/G&gt;
					<B>0.6f</B>
					&lt;A&gt;0.0f&lt;/A&gt;
				&lt;/Color&gt;
				&lt;Near&gt;0.1f&lt;/Near&gt;
				&lt;Far&gt;10000.0f&lt;/Far&gt;
				&lt;Horizontal&gt;45.0f&lt;/Horizontal&gt;
				&lt;Vertical&gt;-1.0f&lt;/Vertical&gt;
			&lt;/Channel00&gt;
		&lt;/Channels&gt;
		&lt;Scene&gt;
			&lt;Name&gt;Scene&lt;/Name&gt;
		&lt;/Scene&gt;
		&lt;Objects&gt;
			&lt;Number&gt;1&lt;/Number&gt;
			&lt;Object00&gt;
				&lt;Name&gt;avatarObject&lt;/Name&gt;
				&lt;Type&gt;gfCal3dObject&lt;/Type&gt;
				&lt;ConfigFile&gt;Marine.xml&lt;/ConfigFile&gt;
				&lt;Motion&gt;Motion&lt;/Motion&gt;
			&lt;/Object00&gt;
		&lt;/Objects&gt;
		&lt;Input&gt;
			&lt;Name&gt;Input&lt;/Name&gt;
			&lt;Type&gt;gfInputGeneric&lt;/Type&gt;
			&lt;Window&gt;window1&lt;/Window&gt;
		&lt;/Input&gt;
		&lt;Motions&gt;
			&lt;Number&gt;1&lt;/Number&gt;
			&lt;Motion00&gt;
				&lt;Name&gt;Motion&lt;/Name&gt;
				&lt;Type&gt;gfMotionHuman&lt;/Type&gt;
				&lt;Input&gt;Input&lt;/Input&gt;
				&lt;Position&gt;
					&lt;X&gt;-27.0f&lt;/X&gt;
					&lt;Y&gt;0.45f&lt;/Y&gt;
					&lt;Z&gt;-1.0f&lt;/Z&gt;
					&lt;H&gt;0.0f&lt;/H&gt;
					<P>0.0f</P>
					&lt;R&gt;0.0f&lt;/R&gt;
				&lt;/Position&gt;
				&lt;Tracker&gt;RifleTracker&lt;/Tracker&gt;
				&lt;FlipJoystick&gt;True&lt;/FlipJoystick&gt;
				&lt;WalkingSpeed&gt;2.0f&lt;/WalkingSpeed&gt;
				&lt;RunningSpeed&gt;4.0f&lt;/RunningSpeed&gt;
				&lt;WalkRunThreshold&gt;95&lt;/WalkRunThreshold&gt;
				&lt;RotationInterval&gt;50&lt;/RotationInterval&gt;
				&lt;GlanceInterval&gt;50&lt;/GlanceInterval&gt;
				&lt;SideStepInterval&gt;75&lt;/SideStepInterval&gt;
				&lt;RotationVelocity&gt;3.5f&lt;/RotationVelocity&gt;
				&lt;StepUpHeight&gt;0.6f&lt;/StepUpHeight&gt;
			&lt;/Motion00&gt;
		&lt;/Motions&gt;
```

```
<Trackers>
	<Number>2</Number>
	<Tracker00>
		<Name>HeadTracker</Name>
		<Port>1</Port>
		<Station>0</Station>
		<Scale>
			<X>1.0f</X>
			<Y>1.0f</Y>
			<Z>1.0f</Z>
			<H>-1.0f</H>
			<P>1.0f</P>
			<R>-1.0f</R>
		</Scale>
	</Tracker00>
	<Tracker01>
		<Name>RifleTracker</Name>
		<Port>1</Port>
		<Station>1</Station>
		<Scale>
			<X>1.0f</X>
			<Y>1.0f</Y>
			<Z>1.0f</Z>
			<H>-1.0f</H>
			<P>1.0f</P>
			<R>-1.0f</R>
		</Scale>
	</Tracker01>
</Trackers>
<Players>
	<Number>1</Number>
	<Player00>
		<Name>GoodGuy</Name>
		<Type>gfPlayer</Type>
		<Motion>Motion</Motion>
		<VisObject>avatarObject</VisObject>
		<Weapon>M16</Weapon>
	</Player00>
</Players>
<Observers>
	<Number>1</Number>
	<Observer00>
		<Name>MainObserver1</Name>
		<Channel>mainChannel1</Channel>
		<Scene>Scene</Scene>
		<Environment>environment</Environment>
		<Position>
			<X>0.0f</X>
			<Y>1.1f</Y>
			<Z>0.0f</Z>
			<H>0.0f</H>
			<P>0.0f</P>
			<R>0.0f</R>
		</Position>
		<TetherOffset>True</TetherOffset>
```

```
                    <SetLookAtObject>NULL</SetLookAtObject>
                    <SetLookAtMode>GFOBS_LOOKAT_NONE</SetLookAtMode>
                    <TetherMode>GFOBS_TETHER_FIX_PCS</TetherMode>
                    <TetherPlayer>GoodGuy</TetherPlayer>
                    <Tracker>HeadTracker</Tracker>
               </Observer00>
          </Observers>
     </System>
```

Figure 20. Example of system.xml file

b) Scenario File

The scenario file encompasses settings that are specific to different scenarios and vary from one scenario to the next, vice settings that remain constant for the application. This includes the actual objects (i.e. models) being placed in the simulation, the lighting and skybox settings, the environment configuration, and the actual targets that are required in the scenario. The example below shows a scenario file—note that as in the system file, the number field change be changed to only include the first two targets even though there might be a list of 50 targets.

```
     <Scenario>
          <Objects>
               <Number>1</Number>
               <Object00>
                    <Name>townObject</Name>
                    <Type>gfObject</Type>
                    <Dataset>
                         <Name>townDS</Name>
                         <File>MOUT_2.6-6-tri_007.flt</File>
                    </Dataset>
               </Object00>
          </Objects>
          <Lights>
               <Number>0</Number>
               <Light00>
                    <Name>light1</Name>
                    <Environment>environment</Environment>
                    <Position>
                         <X>-2.0f</X>
                         <Y>2.0f</Y>
                         <Z>-8.0f</Z>
                         <H>0.0f</H>
                         <P>0.0f</P>
                         <R>0.0f</R>
                    </Position>
                    <AmbientColor>
                         <R>0.8f</R>
                         <G>0.8f</G>
                         <B>0.8f</B>
```

```xml
				</AmbientColor>
				<DiffuseColor>
					<R>0.9f</R>
					<G>0.9f</G>
					<B>0.9f</B>
				</DiffuseColor>
				<SpecularColor>
					<R>0.9f</R>
					<G>0.9f</G>
					<B>0.9f</B>
				</SpecularColor>
			</Light00>
		</Lights>
		<Environment>
			<Name>environment</Name>
			<Shadows>False</Shadows>
			<Fog>
				<Enable>True</Enable>
				<Mode>GF_FOG_EXP2</Mode>
				<Density>0.01f</Density>
				<Near>1.0f</Near>
				<Far>10000.0f</Far>
				<Red>0.5f</Red>
				<Green>0.5f</Green>
				<Blue>0.75f</Blue>
			</Fog>
		</Environment>
		<SkyBox>
			<Name>skyBox</Name>
			<Environment>environment</Environment>
			<FrontFile>cqbFront_512.bmp</FrontFile>
			<BackFile>cqbBack_512.bmp</BackFile>
			<RightFile>cqbRight_512.bmp</RightFile>
			<LeftFile>cqbLeft_512.bmp</LeftFile>
			<TopFile>cqbTop_512.bmp</TopFile>
			<BottomFile>cqbBottom.bmp</BottomFile>
		</SkyBox>
		<Targets>
			<Number>1</Number>
			<Target00>
				<Name>OpFor00</Name>
				<FileName>eqp_misc_target_opfor01.3DS</FileName>
				<ID>1</ID>
				<Position>
					<X>9.262337719</X>
					<Y>0.75</Y>
					<Z>-6.132306352</Z>
					<H>-90.0f</H>
					<P>0</P>
					<R>0</R>
				</Position>
				<Scale>
					<X>0.01875f</X>
					<Y>0.01875f</Y>
					<Z>0.01875f</Z>
```

```
                    </Scale>
                    <Animation>
                        <EndingY>0.0f</EndingY>
                        <FrameStep>5</FrameStep>
                        <TotalTime>0.5</TotalTime>
                    </Animation>
                </Target00>
            </Targets>
        </Scenario>
```

Figure 21. Example scenario file

c) *Character Definition File*

The character definition file defines the individual characters placed into the environment. The settings in this file are defined for specifying the skeleton, mesh, and textures used for the character and the animation used to for that character.

```
<model>
    <scale>0.025</scale>
    <skeleton>Marine_skeleton.csf</skeleton>
    <animation>
        <type>state</type>
        <animation_name>idle</animation_name>
        <transition>11</transition>
        <file>Marine_standing.caf</file>
    </animation>
    <animation>
        <type>state</type>
        <animation_name>walk</animation_name>
        <transition>15</transition>
        <file>Marine_walk.caf</file>
    </animation>
    <animation>
        <type>state</type>
        <animation_name>run</animation_name>
        <transition>15</transition>
        <file>Marine_run.caf</file>
    </animation>
    <animation>
        <type>state</type>
        <animation_name>sidestepleft</animation_name>
        <transition>15</transition>
        <file>Marine_sidestep_left.caf</file>
    </animation>
    <animation>
        <type>state</type>
        <animation_name>sidestepright</animation_name>
        <transition>15</transition>
        <file>Marine_sidestep_right.caf</file>
    </animation>
    <mesh>
        <file>Marine_cover.cmf</file>
        <file>Marine_head.cmf</file>
```

```
                    <file>Marine_body.cmf</file>
            </mesh>
            <material>
                    <file>Marine_material.crf</file>
                    <file>Marine_material.crf</file>
                    <file>Marine_material.crf</file>
            </material>
    </model>
```

Figure 22. Example Character definition file (Marine.xml)

As is shown by the code snippet above and the example XML files, the addition of XML to libGF provides a very easy and structured way to modify an application's system, scenario and character settings without the added necessity of having to recompile the application after each change. Modifying external data files, that are in a well-structured and readable format, greatly increases the flexibility and dynamic nature of any application, especially a virtual environment application where the necessity to rapidly change a scenario exists.

4. Future Work

The limitations of the XML API are because of the library chosen to integrate into libGF. Switching to a more robust library would remove the artificial restrictions placed on the file format requiring the XML entities to be in a sequentially numbered format, as seen in Figure 18, which forces each entity name to be a unique identifier. This restriction is not an XML limitation but, one of the specific library integrated—a more robust library would allow the formatting to be as seen in Figure 17.

Removing the sequentially numbered entity restriction would also help to increase the ease of writing XML files out from within the application. Currently, a counter must be maintained for each portion of the data tree that is to be written. Without the unique entity requirement, the counter would not be required and the writing of data would be more robust.

C. AGENTS

1. Motivation

In many instances, military training suffers because there are not enough personnel to effectively simulate an exercise. Squad training, done correctly, requires a full squad, with three four-man fire teams and a squad leader. Shortchanging the number of personnel can lead to negative training transfer in which squad members know what formations should look like with the wrong number of personnel and will react accordingly even when in a situation where the right number of personnel are present.

Ways to train around a shortage of personnel typically include training as though those personnel are in fact a part of the exercise (or are notional), training smaller units than desired, or simply training incorrectly with a shortage of personnel. Other training methods can be used in place of or in addition to live training, such as tactical decision games, where individuals or small units are asked to think as though they were a battlefield unit, are given a situation and a mission, and are asked to provide a solution or at least a first step toward solving the problem and completing the mission. However, other methods that remove the live aspect of training do not provide the same training or effectiveness as live training. Putting feet on the ground and walking the solution to a problem can have a very different impact from talking through the solution.

So, given a shortage of personnel, one way to make up for those personnel is to provide training in a virtual environment with autonomous agents simulating the additional persons needed. But agents have advantages and drawbacks. The first obvious advantage is that personnel shortage problems are solved. Another advantage is that if there is a need for a "fill-in" person who is needed to perform a task and then leave or to perform a task repetitively, such as a mechanic, a clerk, etc., that does not need to be controlled by a real person, an agent can be used to simulate that person. The main disadvantage, however, is that agents are not real people. While agents can be written to move in a very realistic manner, they are not real, and generally, that is noticeable. However, agents that react to stimulus can be written to act as nearly real as possible, and can provide at least the semblance of realism in a training simulation that might otherwise not.

2. Implementation

Agents in libGF were implemented with two basic premises in mind: 1) all agents will use waypoints when moving around the scene under normal, non-stimulated circumstances, and 2) agents will use a gradient method when deciding what to do or where to go next. In addition to these two basis premises, the implementation needed to account for instances where the agent would handle immediate responses, such as seeing an enemy and moving toward him.

a) Waypoints

Waypoints are locations that the agent is allowed to move to and from under normal circumstance. Waypoints and the corresponding paths between them can be drawn as a complete graph, where the waypoints are vertices of the graph, and paths from each waypoint to all other waypoints represent the edges. In this manner, waypoints can be placed at varying intervals, as sparse or dense as is needed, in order to allow the agent travel over whatever areas the user wants. The fact that waypoints can be placed as sparse as desired saves the computational complexity and the memory requirement of placing a grid of waypoints uniformly over an area, but it places a heavy burden on the user to ensure that there is a path from every waypoint to every other waypoint. This task must be done manually, and, at present, libGF does not incorporate a level editor to create an environment that incorporates automatic or simple waypoint placement. All waypoint information is read into the application and used through the gfWaypointSet class.

b) Gradients

Gradients, in an abstract form, are a means by which autonomous agents can make a choice by finding a numerically superior (or inferior) option across a selection set. This is generally implemented through a vector or matrix of scalar values which can be added to or subtracted from based on input. The agent can then search the gradient values for the largest (or smallest) value. Gradients are implemented in libGF by a one-dimensional array of vectors that each contain gradient information about a specific waypoint. A gradient value is incremented as the desire to go to that gradient's corresponding waypoint increases; likewise, a gradient value is decremented as the desire to go to a gradient's corresponding waypoint decreases. gfAgentGradientMgr is the class that holds the gradient values and adds to and subtracts from those values based on stimulus—when a space is encountered where no enemy is encountered by an agent, the space is considered 'cleared', and the desire to return to that space diminishes immediately, then increases slowly over time to account for the possibility of an enemy having later entered the space. While the additions and subtractions to gradients are arbitrarily based on application need, the current implementation of gfAgentGradientMgr is currently fixed to provide gradient stimulus commensurate with a human character clearing a building.

c) Pathing

The ability to know which waypoint to go to (via gradients) is not enough information to get there. The agent may decide he wants to go to a different room or building, based on the fact that he has not been there in some time, but if there are walls or obstacles between the waypoint he is currently at or near and the waypoint that he wants to go to, then a path must be determined to get him to his destination, and that path must respect the movement requirements of the agent—in the case of human character agents, the path must not take the agent through walls.

In order to be able to find a path from one waypoint to any other quickly, a member of the class gfAgentPathfinder is created when an agent is instantiated. This member takes in the position of all waypoints and uses Kruskal's algorithm for finding the shortest path from each waypoint to each other waypoint, assigning an infinitely large weight to those direct paths from one waypoint to another with an obstacle between them. (using a line-of-sight utility function which tells the algorithm if an obstacle exists between the two points in the virtual space) gfAgentPathfinder preprocesses this information and stores it in a two dimensional array, so that as long as the agent knows what its destination waypoint is and what waypoint it is at, the it can always find the next waypoint to go to directly to reach the destination. In addition to providing fast pathfinding, this method also provides the flexibility in that an agent does not ever need to know the entire path to get to its destination, it only needs to know where it is and where it is going. This becomes a key flexibility issue in returning to and continuing on a path when accounting for off-path actions such as the immediate responses discussed next.

d) Immediate Responses

In addition to the need for an agent to be able to traverse all of its waypoints, there will be times when the agent needs to respond to stimuli by moving off the standard set of paths created by waypoints. In these instances great care must be taken to ensure that the agent cannot put itself in a position where it is isolated or trapped, and a means must always be available to return to the set of paths created by the waypoints. The agent cannot be allowed to roam around freely, as it knows little about its surroundings; a clear and distinct set of rules must be enforced in order to ensure that the agent can return to its original course as needed.

The gfAgent API allows the agent to leave its path only when it sees a member of its 'goodGuys' list. (gfAgent was initially developed with the specific intent to create opposing force agents which would clear rooms looking for good guys, or the Marine participants to the application) When an agent is in direct line-of-sight to a Marine that is a member of its 'goodGuy' list, it leaves its path to go to the Marine. It will continue in this manner until it reaches the Marine or it no longer sees him. If the agent no longer sees the Marine while not on its predetermined path, it continues to the last place where it saw the Marine (such as the Marine's position prior to going around a corner), and if it then sees the Marine again, it continues to move toward him; however, if the agent cannot see the Marine from the Marine's last known position, the agent goes to the nearest waypoint, adjusts his gradients so that he will continue searching in the general vicinity of the Marine's last known position, and otherwise goes back to basic room clearing.

gfAgentActionMgr provides this set of rules through a complicated set of if statements in its update() function. This function is the central point of the gfAgent API and determines whether the agent will chase after a Marine it sees or continue on its course. This provides the agent with a set of rules, and as long as care is taken not to let the agent reach a position where he cannot see a waypoint, the rules will always allow the agent to continue his search. Similar rule sets could also be made for different responses to stimuli; in the same situation of opposing forces and Marines, where the agent is the opposing force, there could be the need for a set of rules that allow the agent to avoid the Marine, to follow the Marine at a distance, or to go to him and then leave. Such similar sets of rules could be separated into individual functions, as could the set of rules built into gfAgentActionMgr::update(), such that multiple sets of rules could be evaluated.

e) Agent Motion

gfMotionAgent is the class which controls the movement of an agent—specifically, his position, his rotation, and his animation. Because our primary focus was on character animation, and because all of the agents produced thus far are human, gfMotionAgent is specific to human motion and is very similar to gfMotionHuman, with the difference being in that the input is being generated within the application as opposed to being generated by an input device, such as a joystick. gfMotionAgent could easily be made generic, however, in order to allow derived agent motion models for various types of agents. Examples would include gfMotionAgentHuman, gfMotionAgentAirplane, gfMotionAgentHelo, gfMotionAgentVehicle, etc.

f) Putting it all Together

gfAgentPlayer is the overarching interface to the gfAgent API, and it is the class that glues the API together. In typical scene graph engine format, some of the gfAgent information is passed in by creating members of encapsulated classes and passing those members to members of their encapsulating classes, thus creating a tiered building architecture. The architecture of the gfAgent API can be seen is displayed in Figure 23.

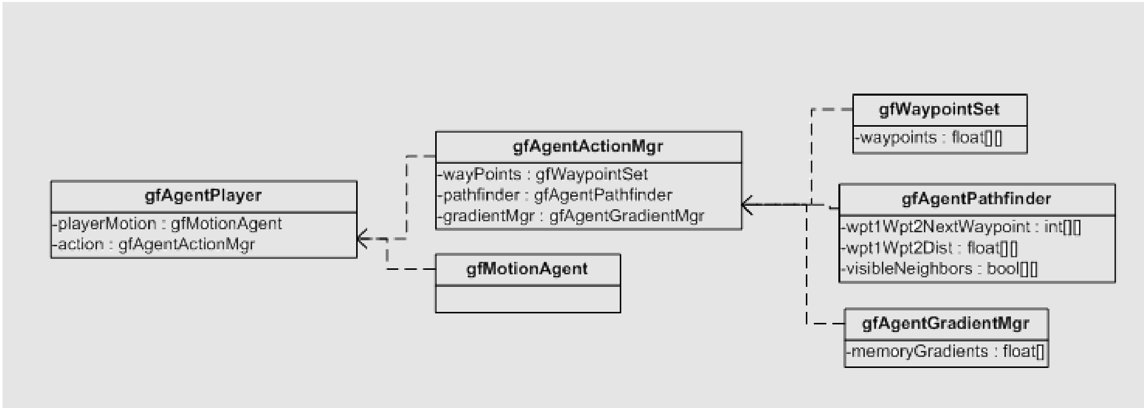


Figure 23. gfAgent API tiered architecture

g) Pathematics

It is worth mentioning the similarity between the gradient-waypoint method of agent manipulation written into the gfAgent API and the pathematics method of routing for autonomous agents written by Alex J. Champandard. The pathematics algorithm is a more robust algorithm that also seeks to use a gradient method by which to move agents intelligently. Though gfAgent was not a product of pathematics, the research done by Alex Champandard is worthy of careful review in extending the gfAgent API.

3. Application

a) How to Create a Set of Waypoints

In order to use the agent capabilities of the gfAgent API, the user has to first create a set of waypoints. Figure 24 shows an example waypoint set in the XML format expected by gfWaypointSet. The reader may note the addition of the path and numpathpoint tags, which are partially implemented into the gfAgent API in order to allow agents to move in a simple path (continuously) through the waypoints.

```
<waypointset>
	<numwaypoints>4</numwaypoints>
	<waypoints>
		<waypoint0>
			<x>-2</x>
			<y>0.2</y>
			<z>0</z>
			<h>0</h>
			<p>0</p>
```

```
                    <r>0</r>
                </waypoint0>
                <waypoint1>
                    <x>-2</x>
                    <y>0.2</y>
                    <z>-5</z>
                    <h>0</h>
                    <p>0</p>
                    <r>0</r>
                </waypoint1>
                <waypoint2>
                    <x>-7</x>
                    <y>0.2</y>
                    <z>-5</z>
                    <h>0</h>
                    <p>0</p>
                    <r>0</r>
                </waypoint2>
                <waypoint3>
                    <x>-7</x>
                    <y>0.2</y>
                    <z>0</z>
                    <h>0</h>
                    <p>0</p>
                    <r>0</r>
                </waypoint3>
            </waypoints>
            <numpathpoints>4</numpathpoints>
            <path>
                <pathpoint0>0</pathpoint0>
                <pathpoint1>3</pathpoint1>
                <pathpoint2>1</pathpoint2>
                <pathpoint3>2</pathpoint3>
            </path>
        </waypointset>
```

Figure 24. a gfWaypointSet waypoints file, WaypointSet.xml

In order to use the waypoints from Figure 24 in the application, a gfWaypointSet must be created that reads the waypoint information from the XML file, as shown in Figure 25.

```
char waypointsFile[256];
gfGetFullFileName("TestWaypointSet.xml", waypointsFile);  //gets the full path name
                                                          //of the file, puts it in the array waypointsFile

gfWaypointSet *wayPoints = new gfWaypointSet(waypointsFile);
```

Figure 25. Creating a gfWaypointSet

b) How to Create an Agent

The creation of an agent involves the creation of a gfAgentPlayer, which is the interface to the API. Before creating a gfAgentPlayer, the programmer must first create a starting position for the agent (a gfPosition), the waypoints (a gfWaypointSet), and a list of 'good guy' names (an array of character pointers) for the agent to potentially seek out. An example of creating a gfAgentPlayer is shown below, in Figure 26.

```
//create the gfPosition for where the agent is to start.  This must be within line-of-sight
//  of a waypoint
gfPosition *agentPos = new gfPosition(2.0f,  0.200000f,  0.0f, 180.0f, 0.0f, 0.0f);

//the list of names of 'good guys' that the agent should look for and go to if he sees
//  one; this must be the character array names given to those gfPlayers that the agent
//  is to look for
char *goodGuyNames[] = {"GoodGuy"};

//the fourth parameter is the number of names provided in the fifth parameter (or the
//  number you want to use; if there are 3 names, and you put a 2 in the fourth
//  parameter, the agent will only look for the gfPlayers associated with the first two
//  names in goodGuyNames
gfAgentPlayer *agentPlayer = new gfAgentPlayer("agentPlayer", wayPoints,
                                               agentPos, 1, goodGuyNames);

//avatarObject is the gfObject, (in this case a gfCal3dObject) already created to be the
//  visual representation of the agent
agentPlayer->AddVisObj(avatarObject);
```

Figure 26. Creating a gfAgentPlayer

Because there is currently only one available motion model for agents (gfAgentMotion), the motion model is created within gfAgentPlayer, keeping the user interface simple by removing the need for the user to know anything about the motion model. When the user adds the visible object to the agent, the agent also makes the motion model a notifier of the visible object, so that the visible object will receive the gfSystem::SendNotify() notifications it needs in order to be positioned by the motion model.

4. Future Work Specific to the gfAgent API

There are several future issues that should be addressed if gfAgent is further developed. Two of these issues are code revisions that have not yet been completed. gfAgent needs the scene geometry to be drawn prior to the preprocessing of waypoint paths, so that line-of-sight determinations between waypoints will account for walls and geometry. Currently, the way the API is written, this usually happens, but not always. gfAgentPathfinder needs to ensure that gfSystem:: UpdateSystem() has been called at least once, to ensure that the geometry is available. In addition to ensuring the validity of paths, code needs added to be able to put agents on a set path, as opposed to allowing rule-based movement. This gives the programmer the ability to script basic movement for such simple program additions as character walk-throughs, and the creation of non-responsive crowds. Paths are already a part of the XML file (Figure 24), and are read into gfWaypointSet. Code needed would give gfAgentActionMgr the ability to distinguish and account for agents that move once or continuously through a path.

All other future work lies in the need to make the gfAgent API more generic, so that it can be used in different scenarios and for different object genres. For instance, if an aircraft agent were desired, the differences needed in the API would include creating a different set of immediate responses (specific to aircraft), a different gradient algorithm (a desire to move toward a waypoint based on a different set of stimuli than with a human genre), and a different motion model. The solution to making the motion model generic is to make gfAgentMotion an abstract class, pass in an enumerated value to the gfAgentPlayer constructor as to which genre was being used, and use the concrete superclass of gfAgentMotion (such as gfAgentMotionAirplane) specific to that genre. Using the same enumerated input to gfAgentPlayer, gfAgentGradientMgr could use different gradient modifier rules also based on genre, and gfAgentActionMgr could also use different genre-based rules to allow for immediate response issues. The structure of the gfAgent API is organized and general enough to allow for needed future flexibility.

D. NETWORKING

1. Motivation

Part of the idea behind creating an architecture to easily create virtual reality training environments is to have the ability for several users to network together and experience a shared virtual environment. We wanted to be able to create a virtual reality training simulation where a fire team of four Marines, or even a squad of thirteen Marines, can get together and practice the procedures required for room and building clearing by simply networking laptops together while deployed. In the past, the vast majority of MOUT training has been accomplished in actual mock-up buildings requiring time, resources, and money. And even with this training, there is still downtime, without any procedural training, while the Marines are deployed on ships. With the creation of a scalable, easily networkable, and deployable virtual reality training system, the downtime can be turned into worthwhile room and building clearing procedural training, while deployed. This was the motivation for adding networking capability to libGF.

Because this was the first iteration, or attempt, at the architecture, a choice had to be made on the extent of the networking capability. There are several networking protocols and simulation interchange schemes to chose from including, but not limited to: Transmission Control Protocol / Internet Protocol (TCP/IP) and User Datagram Protocol (UDP) as the networking protocol, Client/Server and Peer-to-Peer for the network architecture, and DIS and HLA as the simulation interchange protocol. Since the architecture was designed to create small-scale virtual environments, but still have the ability to network sixteen or so users together and not necessarily provide the ability to interchange with other, already existing simulations packages, the decision was made to first implement a peer-to-peer scheme using UDP and not incorporate HLA or DIS. The benefit of using a peer-to-peer architecture is that it has "the advantage of minimizing latency for packet delivery by sending packets via the shortest path from source to destination." (Singhal and Zyda, 1999, p. 243) UDP was chosen vice TCP because the need for a guarantee of packet delivery did not exist, and using UDP provided a "simple best-effort delivery semantic on transmitted data packets" (Singhal and Zyda, 1999, p. 7); missing the occasional packet did not limit the functionality of the environment and could be overcome when the next packet was received. The addition of HLA is to be added at a later date and is discussed more in the future work section.

2. Implementation

The choice as the underlying library to use for networking was partially based on the additional need for a library that could handle interface input easily. For that reason, the library used to integrate networking capability into libGF was DirectPlay® in the Microsoft® DirectX® SDK[17]. DirectPlay® provides both peer-to-peer and client/server architectures; as previously stated because of the relatively small nature of the environment in mind, we decided to go with the peer-to-peer setup.

The first step in integrating networking was to create a basic networking class that would handle the DirectPlay® interface and basic connections between computers. This base class served to establish the initial connection, determine which machine was acting as the "host" machine for the session, and which were acting as simply a "peer." Even though the conventional aspect of peer-to-peer architectures are host-less, DirectPlay® incorporates a host to act as the session controller—the host handles the requests from all peers requesting to join the specific session in question. As discussed in the Microsoft® DirectX® (C++) SDK Help, the session host is responsible for managing the session, including:

- Managing the list of session members and their network addresses
- Deciding whether a new user is allowed to join the session.
- Notifying all members when a new user joins the session, and passing them the new user's address.
- Providing new users with the current game state
- Notifying all users when a user leaves the session

[17] Microsoft, DirectX, and DirectPlay are registered to Microsoft Corporation – http://www.microsoft.com

Once each peer has requested to join the session, is granted permission to join by the session host, and has joined the new session, the peer will automatically receive all messages sent from the other peers on the network. After the individual computer has joined a session, it is responsible for sending out the appropriate messages and handling any new network traffic. The base networking class—gfNetwork—only handles the basic connection, creating players, destroying players, and shutdown messages. Any additional network traffic needs to be handled by a network class derived from the gfNetwork base class.

Since our primary focus was providing the addition of character animation to the libGF library, we derived a new class from the gfNetwork base class, called gfCal3dNetwork. This new class handles network messages geared more to handling the character animation aspects in an application. Specifically, upon starting an application and being accepted by the session host to join the session, a packet is sent out to create a new player—a new gfCal3dPlayer. Then every ten seconds a new state packet is sent out to ensure that the initial state of the avatar is known to all peers on the network. The reason this packet is continually sent is to ensure any new peers joining the session are updated to know about all existing avatars in the environment—in case any players join the session after it is initially started. When a player state packet is received, from a player that has not already been initialized, the peer receiving the state packet will initialize the new player and add the avatar to the environment. From then on, any packets pertaining to that newly created player will result in updates being made to that player on all peers throughout the network. The packet types that are sent and received, and the appropriate actions that are taken based on the specific packet, are shown in the following table:

Packet type	**Action**
GFPACKET_TYPE_POSITION	**Send**: As the avatar position changes, the new position is updated and a new position packet is sent over the network.

	Receive: When a new position packet is received, the position of the respective avatar is updated.
GFPACKET_TYPE_PLAYER_STATE	**Send**: The state of the avatar is sent out every ten seconds (for reasons already discussed). This state includes the file name describing the specific character to add and the initial position to establish for that character. **Receive**: Upon receiving the player state packet for the first time, the character file is read, the new character loaded, and the initial position applied to the new avatar.
GFPACKET_TYPE_PLAYER_ACTION	**Send**: As the avatar action changes based on the input from the input interface, the current animation of the avatar changes and must be updated on the machines throughout the network. In order to do that, an action packet is sent containing the name of the animation being currently played for that player. **Receive**: The new action packet, when received is passed to the player and the new animation is started for that player. Via this packet, players throughout the network are continually displaying the correct animations as they are being controlled on the machine that owns that character.
GFPACKET_TYPE_FIRE	**Send**: When the weapon associated with the local player is fired, a fire packet is sent so that the weapons of the remote players are fired. Targets hit and a weapon firing sound are seen and heard on remote machines as they are occurring on the local machine. **Receive**: A firing packet received causes the weapon for the respective player to be fired and a bullet displayed, in the appropriate location, if any objects are hit by that weapon.

Table 1. gfCal3dNetwork Packet Descriptions

Any new packets requiring passing and handling through the network can be easily added to the network class, or a new network class can be derived. A packet is simple a new class that is derived from the gfBasePacket class and contains the new data to be passed. An example would be the action packet that was necessary for passing character animations from a local player to remote players in the network, as shown below.

```
class gfPlayerActionPacket: public gfBasePacket
{
public:
   gfHumanActions action;
   char command[50];
};
```

Figure 27. Example of a new network packet

In this case, the gfPlayerActionPacket needed additional information to be passed to allow animations by the player. The gfBasePacket class already contains the type and size of the packet, so the additional information of the specific action and command string were added to a new derived packet class. Any additional packets can be created and added in the same manner.

3. Application

The use of the gfCal3dNetwork class in any new application is accomplished by following a few key steps. A new network object must first be instantiated in order to have objects able to send and receive appropriate packets.

```
gfCal3dNetwork *network;

if(networkMode == HOST) {
        network = new gfCal3dNetwork("Local", NULL, true);
}
else if(networkMode == CLIENT) {
        network = new gfCal3dNetwork("Local", networkServerStr, false);
}
```

Figure 28. Creating a new network object

Once a network object has been instantiated, objects needing to send and receive network packets must subscribe to the network—add it as a notifier. This is accomplished by:

```
// This adds a notifier for the motion model to the network allowing messages
```

```
// to be sent from the motion model to the network object
network->AddNotifier(motion);

// Likewise to have network packets received by another object, simply add a
// notifier for the network to the object interested.
weaponMgr->AddNotifier(network);
```

Figure 29. Adding message passing/receiving capability

Before sending a data packet over the network, set the data inside the motion model and use the SendNotify command. SendNotify is the mechanism that allows messages to be passed between objects once they have been added as listeners via the AddNotifier method. Then, inside the derived network class, set the packet type and size and send the packet.

```
// In the motion model set the data and then send a notify message – i.e. in
// this case an new action of ‘Run’ is being set to play the run animation:
currentState = eB_RUN;
currentStateStr = "Run";
data->SetAction(currentState);
SendNotify(currentStateStr, data);

// In the derived network class, when the data is received from the motion
// model, create a new action packet, set the type and size, and then send the
// packet to the network
gfPlayerActionPacket *packet = new gfPlayerActionPacket();
packet->mType = GFPACKET_TYPE_PLAYER_ACTION;
packet->mPacketSize = sizeof(gfPlayerActionPacket);
packet->action = ((gfHumanRefData*)message->getData())->GetAction();
sprintf(packet->command,"%s", command.getString());

Send(packet);
```

Figure 30. Sending a data packet from the motion model to the network

When other computers on the network receive the packet, a test is done to determine the type of packet received, and then the appropriate action is taken.

```
// Determine which kind of packet is received and then calling the
// appropriate method to handle the received packet:
void gfCal3dNetwork::ReceiveMessage(gfMessageData *data)
{
	switch (data->mPacket->mType) {
	case GFPACKET_TYPE_POSITION:
		//Process a remote player's position
		ProcessPlayerPosition(data);
		break;

	case GFPACKET_TYPE_PLAYER_ACTION:
		//Process a remote player's action
		ProcessPlayerAction(data);
		break;

	default:
```

```
                    printf("Unknown packet type:%d\n", data->mPacket->mType);
                    break;
          }
}

// A packet type GFPACKET_TYPE_PLAYER_ACTION is received,
// get the data and use SendNotify to pass the message to the character
// to play the appropriate animation:
void gfCal3dNetwork::ProcessPlayerAction( gfMessageData *data )
{
           gfPlayerActionPacket *packet = (gfPlayerActionPacket*)data->mPacket;

          static char IDString[32];
          sprintf(IDString, "%x", data->mPlayerID);

          gzRefPointer<gfPlayer> player = gfFindPlayer(IDString);
          if(player) {
                    SendNotify(packet->command, data);
          }
}
```

Figure 31. Receiving a network packet and handling the data

So, with only a few extra steps, a network can be formed and packets sent and received to allow animated characters to be seen by all users on the network. New types of packets can be easily added using the methods shown. The key points are ensuring that the right objects are added as listeners (via the AddNotify method), that the data is sent to the network class (via the SendNotify method), the packet type is set and the packet sent to the network. When a packet is received, the packet type must be determined and then the data contained in the packet is processed.

4. Future Work

As discussed above, the current limitation of the networking API is the inability to communicate with other applications using DIS or HLA. The library is centered on the DirectX® API which provides the necessary networking functionality, but at the same time limits the flexibility of the communication. Using DirectX® for its functionality is with the assumption that communication will be handled by DirectX® and that the joining/leaving of sessions will be enforced by the host.

Switching to a different networking architecture, or removing DirectX® as the underlying framework, would provide a more flexible networking scheme and would remove the requirement of a strict session management. A better implementation would be one of allowing a broadcast of packets to all users subscribed to a broadcast group and the ability to easily switch to sending DIS or HLA packets. The new packet format would facilitate integration with other simulations and bridge the gap between having different applications running with different missions, but be integrated into a common picture—one could be flying a helo mission, a different one could be working with urban vehicles, one application calling for fire support, and the close quarter combat application with animation Marine models clearing buildings—all connected and seeing the common picture.

E. PHYSICS

1. Motivation

A large aspect of the realism of an application—at least for a first person application where the participant is led to believe that he is really a part of the virtual world—lies in the reaction of the environment to participant movement and actions. If a participant in a first person application runs into a wall, he expects his virtual representation to stop at the wall. Similarly, expected results can be defined for virtual movement such as stepping off of a ledge (the expectation of falling), stepping toward a staircase (the expectation that the virtual representation will climb stairs in a way that looks correct), or having an object thrown at or shot at the first person representation (the expectation that the first person representation will be momentarily pushed or bumped by the object being projected).

For a smaller application, or where movement is limited or scripted, these kinds of reactions can be written into code as specific responses to specific inputs, but coding in that fashion removes the flexibility of what can happen, removing the ability to explore new possibilities. In order to allow realistic response to stimulus in the virtual world, some form of physics needs to be implemented into the architecture.

At the very least, to prevent virtual participants from moving freely through walls and falling through floors, collision detection and response needs to be implemented. Collision detection can be viewed in one of two ways; either as a subset of physics or as a separate issue entirely. As long as collision detection and response is directly tied into the physics implementation, which way to view it is inconsequential. However, collision detection ends where realistic response begins; collision detection tells the programmer when and where objects collide. It does not tell the programmer what response to take in relation to that information; that aspect, known as collision response, is most effectively implemented through the use of a physics implementation that provides a realistic response to known collisions.

2. Implementation

a) Use of Existing Technology

Because much study has already been done in the area of collision detection and response and in the area of physics, the authors chose to study existent, open-source collision and physics interfaces in order to determine the best method of producing realism with the greatest amount of code and package reuse. Collision libraries studied included: OpCode, ColDet, SOLID, V-Collide, I-Collide, and QuickCD. In addition to looking into collision detection libraries, the authors chose to look into open-source physics libraries as well; the only physics library studied in depth was the Open Dynamics Engine (ODE)[18], written by Russ Smith.

b) Physics Through Inheritance and Encapsulation

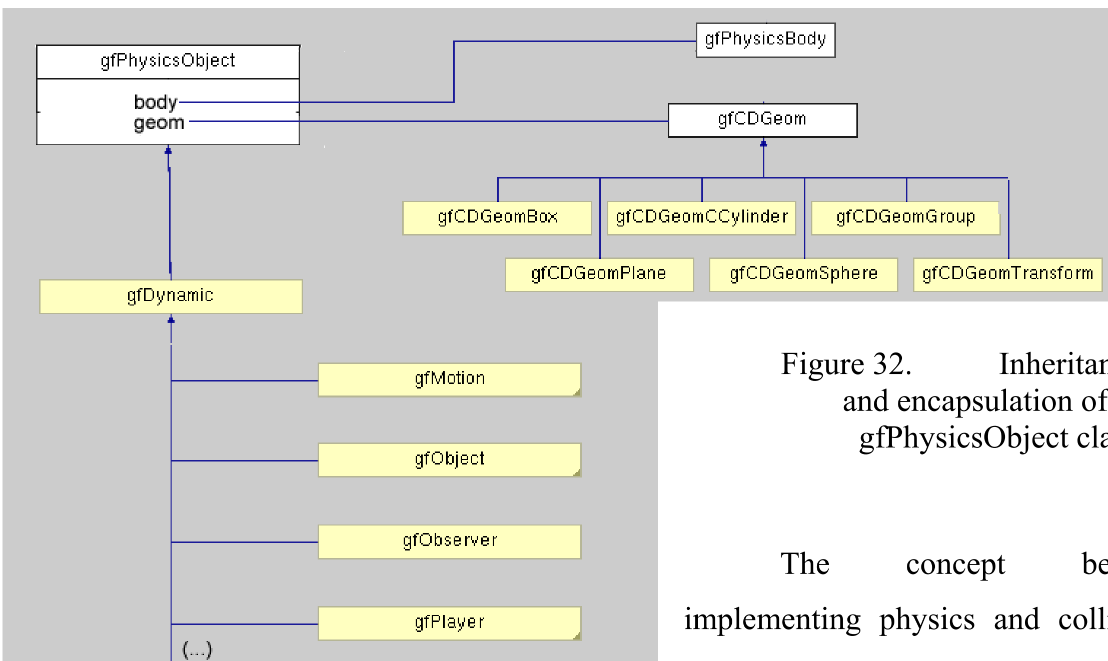


Figure 32. Inheritance and encapsulation of the gfPhysicsObject class

The concept behind implementing physics and collision detection into libGF was to wrap the functionality built into ODE with a thin wrapper that not only took advantage of the optimizations built into ODE, but connected ODE functionality to libGF in a way more consistent with scene graph engine functionality rather than physics SDK functionality. To that end, an attempt was made to separate collision and physics functionality in libGF,

[18] ODE is the Open Dynamics Engine, written by Russ Smith. It is licensed under the GNU LGPL license. http://opende.sourceforge.net

partly because the two are separated functionalities in ODE, but more importantly, because the two functionalities are separate and, therefore, need to be separable. Collision functionality can be found exclusively in gfCDSpace, gfCDGeom, and those classes that derive from gfCDGeom. Physics functionality can be found in gfPhysicsBody and gfPhysicsWorld. Both functionalities are implemented via encapsulation in gfPhysicsObject, which gfDynamic derives from (see Figure 32), such that any class deriving from gfDynamic (there are many) has the capability of having physics properties. This makes those physics properties semitransparent to the user, such that all a user has to do is turn on and turn off functionality from the derived class. Additionally, gfSystem derives from gfPhysicsWorld and gfCDSpace (see Figure 33), so that the functionality of the physics

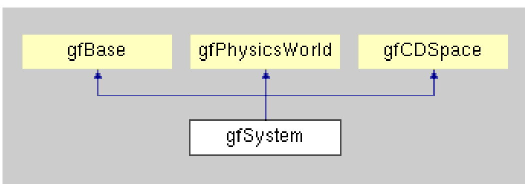


Figure 33. gfSystem inherits gfPhysicsWorld and gfCDSpace functionality

world and the collision detection space can be handled at the system level and be transparent to the libGF end user. When a member of gfSystem is created, a physics world is created in gfPhysicsWorld and a collision space is created in gfCDSpace; these are the spaces where the physics bodies and collision geometries are to be made and placed. The inheritance of public functions by gfSystem allows the programmer to use a handle to a member of gfSystem to add and manipulate those physics bodies and collision geometries.

c) Abstracting Physics Functionality into Core libGF Classes

In addition to direct implementation of classes, several core libGF classes needed functions or functionality added to support physics capabilities. Most importantly, gfSystem, which runs the main system control loop, steps the physics world (which is created in gfPhysicsWorld) ahead by the same amount of time as the system step time. In physics simulations, it is important to keep time steps consistent and small, so that objects do not penetrate or interfere with each other to any large degree; allowing serious interpenetration or interference between objects creates large calculations between the objects and results in ‘explosions’ of the simulation, where objects fly away or just vanish. In order to avoid this possibility, when stepping the physics world ahead by the same time as the libGF cycle, the time stepped on a libGF cycle is divided by a constant physics world step time small enough to prevent physics calculation instability. The physics world is then stepped the number of iterations calculated, in physics world step time intervals, as shown in Figure 34.

```
  int i;
  const static double WORLD_STEP_TIME = 0.01;

  //divide the time since the last system frame by the time each ODE (physics)
  // world step needs to be to get the number of whole world steps we currently need to
take
  int numWorldSteps = (int)(mSysData.mDeltaFrameTime/WORLD_STEP_TIME);


  //step the physics world by the set world step time for the number of
  // iterations as derived above in numWorldSteps
  for (i=0; i < numWorldSteps; i++)
  {
    //account for all collisions at present time
    cdCollide();

    //then step the world, letting objects move in the physics world
    gfWorldStep(WORLD_STEP_TIME);

    //and finally, remove all collision contact points, so the process can be repeated
    emptyContactGroup();
  }
```

Figure 34. gfSystem physics step loop in main execution loop

Of particular note is that the above iterations of physics world step times seems to leave a remainder of time stepped forward in the libGF (visible) world, but not in the physics world. To account for this leftover time, the code in Figure 35 was implemented immediately following the loop of all world step time iterations. However, stepping the physics world by this leftover time made the physics world unstable and caused applications to crash, so the section of code was removed, and stability was restored. No solid explanation for this behavior has yet been found, though the cause may be the variability of the leftover time or the possibility of division by near-zero numbers. Though it is not currently implemented, one way to make up for the leftover time is to save it and add it to the libGF step time (mSysData.mDeltaFrameTime in Figure 35) on the next step.

```
//calculate the leftover time (not a whole world step)
double leftoverWorldStepTime =
    mSysData.mDeltaFrameTime - numWorldSteps * WORLD_STEP_TIME;

//account for all collisions at present time
cdCollide();

//then step the physics world by the leftover time
gfWorldStep(leftoverWorldStepTime);

//and finally, remove all collision contact points, so the process can be repeated
emptyContactGroup();
```

Figure 35. gfSystem physics leftover time step in main execution loop; removed due to instability

In addition to stepping the physics world, additional code needed to be implemented to ensure that members of gfDynamic (which derives from gfPhysicsObject) that had physics turned on were positioned according to their physics world equivalent—otherwise, there is no link between the objects moving in the physics world and objects moving in the visible libGF scene. This needs to be done in two place: a) every cycle (Figure 36) from the main update loop in gfSystem, and b) when objects are specifically positioned, as per Position() in gfDynamic (Figure 37).

```
//(excerpted from gfSystem)

  //iterate through all gfDynamic members
  for (int dynamicNum = 0; dynamicNum < DynamicList->GetNum(); dynamicNum++)
  {
    gfDynamic *getDyn = (gfDynamic *)DynamicList->Get(dynamicNum);

    //the member's physics is enabled, then update the position according to its physics
    //  representative position
    if (getDyn->physicsEnabled())
    {
       getDyn->updatePosition(mSysData.mDeltaFrameTime);
    }
  }


//(excerpted from gfDynamic)

/**update the visual position from the physics position (based on the current
  physics position in the physics world*/
```

```
void gfDynamic::updatePosition(const double deltaFrameTime)
{
   sgMat4 posMatrix = {0.f};
   getPhysicsPosition(posMatrix);
   setVisualMatrix(posMatrix);
   setVisualGfPosition(posMatrix);
}
```

Figure 36. Stepwise update of objects in the scene relative to their physics representation

The code in Figure 36 follows the code in Figure 34 in the main execution loop, so the physics position is first updated by stepping the physics world by the libGF step time, and the visual positions of objects in the libGF scene are then updated according to the position of their respective physics representations. In this manner, objects in the scene—which are not, by themselves, physically based—appear to move or be affected by the forces of physics.

```
/**set the visual scene position and the physics position, given a matrix */
void gfDynamic::Position( sgMat4 srcTransform )
{
   //zero the forces on the body prior to repositioning it
   zeroBody();

   //set the matrix form of the position
   setVisualMatrix(srcTransform);

   //set the HPR form of the position
   setVisualGfPosition(srcTransform);

   //set the position of the physics body to match the visual object
   setPhysicsPosition(srcTransform);
}
```

Figure 37. Repositioning an object, which updates its visual position and its related physics object position

Updating a member's position via gfDynamic::Position() can be done by passing in an HPR representation or a 4x4 transformation matrix, as both representations are stored in gfDynamic. The reader will note that while there is only one matrix representation of a given position and rotation, there are many HPR representations. Whereas gfDynamic::setVisualMatrix(gfPosition *) is capable of preserving correct position and rotation, gfDynamic::setVisualGFPosition(sgMat4) merely chooses the best possibility to maintain similar HPR from previous time step.

3. Application

a) How the System Starts a World and Space

Creation of the physics world and collision detection space, which are individual entities in the Open Dynamics Engine representation, is abstracted through creation of a gfSystem. See Appendix A, SectionB (libGF Quick-Start Guide) for an example of how to create a gfSystem.

b) Setting Global Physics and Collision Parameters

Physics calculations can be performed either faster or more accurately, based on global setting, as shown in Figure 38. In general, when representing many objects, set the physics step type to PHYSICS_SPEED so that the physics calculations do not slow down the application. This does, however, reduce the accuracy of internal physics calculations, so if there is no visible difference between the two settings, leave the step type set to (the default) higher accuracy.

```
        //to make the physics faster

sys->setPhysicsStepType(PHYSICS_SPEED);

        //or to make the physics more accurate

sys-> setPhysicsStepType (PHYSICS_ACCURATE);
```

Figure 38. Setting the physics step type for accuracy or speed

In addition to setting the calculation speed, world gravity can be set, so that gravitational forces can be handled easily. (Figure 39) Note that gravity is set by default to –9.81 (m/s^2) in the y-axis.

```
sys->setGravity(0.0f, -9.81f, 0.0f);
```

Figure 39. Setting gravity for a simulation

c) How to Create a Geometry

The use of collision detection requires that a collision geometry be created to pass to the gfDynamic object. Creation of all of the basic gfCDGeom types is done as shown in Figure 40. gfCDGeom is an abstract class, so the programmer must instantiate a member of a concrete subclass—gfCDGeomBox, gfCDGeomPlane, gfCDGeomSphere, or gfCDGeomCCylinder, which are the basic shape geometries, or gfCDGeomTransform and gfCDGeomGroup, which will be discussed separately. The reader will note that standard behavior is such that geometries can be rotated and positioned; however, a gfCDGeomPlane is an infinite plane and can only be created and destroyed, not moved.

```
gfCDGeom *boxGeom0 = new gfCDGeomBox("boxGeom0", 10.f, 1.f, 10.f);
```

Figure 40. Creating a gfCDGeom

d) *How to Create a Transform Geometry*

When a collision geometry is attached to a physics body (a gfPhysicsBody), the geometry's position becomes that of the body. If the geometry's position is its center and the body's position is a point (which are both the case), this centers the geometry on the body (ex, the single-point physics mass is the center of a sphere; the sphere geometry is centered on that point). In order to use multiple simple geometries to create complex composite geometries or to offset the mass from the center, the user needs to be able to offset the simple geometries from the physics body position. The way to do this is to create a gfCDGeomTransform. The gfCDGeomTransform is passed a gfCDGeom at instantiation, and then an offset position for that geometry. The gfCDGeomTranform member's position and rotation, when added to a physics body, are that of the physics body, but the position/rotation of the collidable geometry (the geometry passed into the transform) is the transform's position/rotation plus the offset.

```
gfCDGeom *ballGeom = new gfCDGeomSphere("ballGeom", 1.3f);
gfCDGeomTransform *ballTransform = new gfCDGeomTransform("ballTransform",
                                                                ballGeom);
gfPosition *ballGeomPos = new gfPosition(0.f, 1.1f, 0.f, 0.f, 0.f, 0.f);
 ballTransform->setOffset(ballGeomPos);
```

Figure 41. Creating a gfCDGeomTransform

e) *How to Create a Group Geometry*

In order to create constructive, complex collision geometries from simpler shapes, the user can create geometry groups that encapsulate multiple geometries.

Because the simple geometries will generally need to be offset from each other and from the center of the physics body, the gfCDGeomGroup is often best used in coordination with the gfCDGeomTransform. Grouping geometries can be done in one of two ways: either by manually adding geometries to a geometry group, or by reading an entire group into a gfCDGeomGroup via an XML file. An example of manually adding geometries to a group is seen in Figure 42, while Figure 43 depicts reading a group from an XML file. Note than geometry groups which are read in from an XML file can still be added to manually.

```
// make a geometry group by manually adding  geometries

//… make simple geometries, then make transforms to rotate and position
//  the geometries where they will need to be in relation to the center of the group…

gfCDGeomGroup *houseGeomGroup = new gfCDGeomGroup("houseGeomGroup”);

houseGeomGroup->addGeom(boxTransform0);   //add a geometry (transform) to the
                                          //  empty group
houseGeomGroup->addGeom(boxTransform1);   //then keep adding until all needed
                                          //  geometry is added
//…
```

Figure 42. Creating a gfCDGeomGroup through manual additions

```
// make a geometry group from an xml file
char houseGeomFile[256];
gfGetFullFileName("houseGeomGroup.xml", houseGeomFile);

gfCDGeomGroup *houseGeomGroup = new gfCDGeomGroup("houseGeomGroup",
                                                  houseGeomFile);
```

Figure 43. Creating a gfCDGeomGroup via an XML file

The format for creating a geometry group XML file is depicted in Figure 44. The top level is <geomgroup>, which is different from all other levels. Note that a group can contain transforms and other groups. Also note that, while transforms may contain individual geometries or groups, they may not contain transforms; this does not generally pose a problem, as groups containing transforms can be positioned in the application.

```
<!—adds 5 transforms of gfCDGeomBox members; three solid walls and one  -->
<!—wall with a doorway in the middle  -->
<geomgroup>
    <numgeoms>5</numgeoms>
    <geom0>
```

```
		<name>wall0transform</name>
		<type>transform</type>
		<offset>
			<x>-4.515</x>
			<y>1.350</y>
			<z>5.301</z>
			<h>0.0</h>
			<p>0.0</p>
			<r>0.0</r>
		</offset>
		<geom0>
			<name>wall0geom</name>
			<type>box</type>
			<xlength>5.666</xlength>
			<ylength>2.700</ylength>
			<zlength>0.318</zlength>
		</geom0>
	</geom0>
	<geom1>
		<name>wall1transform</name>
		<type>transform</type>
		<offset>
			<x>3.552</x>
			<y>1.350</y>
			<z>5.301</z>
			<h>0.0</h>
			<p>0.0</p>
			<r>0.0</r>
		</offset>
		<geom0>
			<name>wall1geom</name>
			<type>box</type>
			<xlength>7.598</xlength>
			<ylength>2.700</ylength>
			<zlength>0.318</zlength>
		</geom0>
	</geom1>
	<geom2>
		<name>wall2transform</name>
		<type>transform</type>
		<offset>
			<x>7.201</x>
			<y>1.350</y>
			<z>-2.500</z>
			<h>0.0</h>
			<p>0.0</p>
			<r>0.0</r>
		</offset>
		<geom0>
			<name>wall2geom</name>
			<type>box</type>
			<xlength>0.294</xlength>
			<ylength>2.700</ylength>
			<zlength>15.921</zlength>
		</geom0>
```

```
    </geom2>
    <geom3>
        <name>wall3transform</name>
        <type>transform</type>
        <offset>
            <x>-7.201</x>
            <y>1.350</y>
            <z>-2.500</z>
            <h>0.0</h>
            <p>0.0</p>
            <r>0.0</r>
        </offset>
        <geom0>
            <name>wall3geom</name>
            <type>box</type>
            <xlength>0.294</xlength>
            <ylength>2.700</ylength>
            <zlength>15.921</zlength>
        </geom0>
    </geom3>
    <geom4>
        <name>wall4transform</name>
        <type>transform</type>
        <offset>
            <x>0.0</x>
            <y>1.350</y>
            <z>-10.301</z>
            <h>0.0</h>
            <p>0.0</p>
            <r>0.0</r>
        </offset>
        <geom0>
            <name>wall4geom</name>
            <type>box</type>
            <xlength>14.696</xlength>
            <ylength>2.700</ylength>
            <zlength>0.318</zlength>
        </geom0>
    </geom4>
</geomgroup>
```

Figure 44. Creating a geometry group XML file

f) Collision Geometry Settings

The only currently implemented setting for collision geometries is the slip coefficient. The slip coefficient is what determines a geometry's friction against other geometries—its slipperiness. Figure 45 shows how to set slip.

```
geom.->setSlip(0.01f);
```

Figure 45. Setting the slip coefficient for a collision geometry

g) How to Create a Body

Whereas collision detection requires a geometry, the use of physics properties (forces acting on a body) requires that a physics body be created to pass to the gfDynamic member. gfPhysicsBody is created as shown in Figure 46. A physics body, by itself, is a point mass that has (internally) an inertial matrix that gives the mass inertial properties. Giving the mass shape is covered in the next section.

```
gfPhysicsBody *boxBody = new gfPhysicsBody("boxBody");
```

Figure 46. Creating a gfPhysicsBody

h) Physics Body Settings

All currently implemented settings for physics bodies deal with mass distribution. Figure 47 depicts methods of setting and manipulating the mass of a physics body.

```
//methods for setting the mass of a body; automatically creates the inertial matrix
void setMassToPoint(float pMass);
void setMassToSphere(float pMass, float pRadius);
void setMassToCCyl(float pMass, int axis, float pRadius, float pLength);
                                  //axis needs to be 0, 1, or 2, for X, Y, or Z, respectively
void setMassToBox(float pMass, float pX, float pY, float pZ);

//methods for setting the offset of the mass from the physics body's point position
void offsetMass(gfPosition *pPos);
void zeroMassOffset();
```

Figure 47. Setting the mass on a physics body

i) Attaching/Detaching a Geometry to/from a Body

Forces can act on a body directly as a point mass, but the application of forces by collision detection (preventing a body from passing through objects such as walls and ground) requires that the collision geometry be attached to the body. Adding the forces of collision detection to the body gives the body form and shape, by putting the body inside the collision geometry. Once connected, a geometry's position is synonymous with the associated body's position; moving one moves the other. Thus, wrapping the physics body such that the center of mass is particularly located inside the geometry requires an understanding of geometry transforms and groups, as already discussed in sections II.E.3.d and e. Because geometry may need to be changed dynamically (i.e., change the shape of an object due to damage), geometry can be both attached to and detached from a body; this allows for the dynamic interchange of geometries to a physics body (or vice versa).

The standard interface for attaching/detaching collision geometries and physics bodies is to set the geometry and the body to the gfDynamic member (see next section). Because geometries and bodies can be created independently, the current implementation of libGF supports the ability to attach geometries directly to physics bodies without the requirement of working through a gfDynamic, but this interface is not depicted here, as it will be deprecated in future use, when geometries and physics bodies will be created internally to gfDynamic members and their public interface hidden.

j) How to Set the Geometry and Body to the gfDynamic Member

Once a gfCDGeom member and a gfPhysicsBody are created, the way to tie them to an object in the scene is to set them as the collision geometry and physics body for a gfDynamic member. The gfDynamic class member (with its inherited functionality from gfPhysicsObject) is the glue that allows for interaction between visible scene objects, collision geometries, and physics bodies. Classes that are usable in the scene, such as gfObject, gfMotion, or gfPlayer, all inherit from gfDynamic. Setting the geometry and physics body to a gfDynamic member not only ties them to the visible scene object, but also ties the collision geometry and physics body together. The gfDynamic public interface for manipulation of collision geometries and physics bodies is described in Figure 48.

```
gfObject *dynamicObject = new gfObject("dynamicObject");

//set the geometry or the body; setting a geometry or body when one is already set
//  will automatically remove the old geometry or body and disable it
dynamicObject->setCDGeom(geom);
dynamicObject->setPhysicsBody(body);

// remove the geometry or the body and leave a null
dynamicObject->removeCDGeom();
dynamicObject->removePhysicsBody();

//methods for getting the member’s geometry or body (to change settings)
dynamicObject->getGeomID();  //geometry can then be retrieved through
                             //  gfFindCDGeom(geomID)
dynamicObject->getBodyID();    //geometry can then be retrieved through
                             //  gfFindCDGeom(geomID)
```

Figure 48. Setting/removing the geometry and body of a gfDynamic member

k) How to Explicitly Enable/Disable Collision Detection/Physics

In order to allow the flexibility of being able to turn physics or collision on or off as necessary, enabling and disabling functions are implemented, so that the user does not have to set and remove geometries and bodies in order to enable or disable their abilities. The interface for enabling or disabling physics or collision is displayed in Figure 49.

```
gfObject *dynamicObject = new gfObject("dynamicObject");

//disable functions
dynamicObject->disableCollision();
dynamicObject->disablePhysics();

//disable functions
dynamicObject->enableCollision();
dynamicObject->enablePhysics();

//boolean functions to determine whether collision or physics are enabled
dynamicObject-> collisionEnabled();
dynamicObject-> physicsEnabled();
```

Figure 49. Enabling/disabling collision detection and physics

l) Collision Detection and Physics Enable/Disable Defaults

In discussing how to set collision geometries and physics bodies to gfDynamic members, the enable/disable defaults should also be discussed, so that the application programmer knows what to expect. When performing a setCDGeom() on a gfDynamic member, collision detection is automatically turned on, and if a physics body is already set, the collision geometry is attached to the physics body. This is true not only when a collision geometry is initially set for the gfDynamic member, but is also true if the members collision geometry is switched out. Likewise, when performing a setPhysicsBody() on a gfDynamic member, physics properties are automatically turned on, and if a collision geometry is already set, the geometry and body are attached to one another. It is also true that when a geometry or body is removed from a gfDynamic member, the geometry and body are detached from each other and the collision or physics, respectively, is disabled.

m) gfDynamic Member Physics/Collision Configurations

Because physics and collision are implemented into libGF independently, there are four possible physics configurations a gfDynamic member can have:

- Collision detection (without physics) can be implemented by setting the gfCDGeom to a member and not setting the gfPhysicsBody, or by disabling the physics body. For non-moving (static) objects, this creates a collidable object in the scene that cannot be moved. This configuration is useful for walls, buildings, ground, and any objects that should not be moved by physics forces (though the objects, and their respective collision geometry, can be manually repositioned).
- Physics (without collision) can be implemented by setting the gfPhysicsBody to a gfDynamicmember, but not a gfCDGeom. In this case, all forces are directly added to the member, and care needs to be taken to account for gravity or any other forces that destabilize the member's position and orientation.
- Collision detection and physics can both be implemented at the same time, by setting both a gfCDGeom and a gfPhysicsBody to the gfDynamic member; this is the standard physically based model which moves in the environment based on all forces, both internal and external, to include collisions with other physical bodies in the environment.
- Neither collision detection nor physics are implemented. No gfCDGeom and no gfPhysicsBody are set to the gfDynamic member, which is moved through the environment by device input and non-physically based algorithms. Movement is sterile and easy to control, since the lack of variable forces and collision-based restraints prevents unexpected movement of objects in the scene. This method is useful for absolute control over a gfDynamic member.

n) How to Set a Ground Plane

The current libGF physics implementation does not support creating collidable triangle meshes, so, currently, the only way to create a ground plane is to add a gfCDGeomPlane to the collision space. This can be accomplished in one of two ways (see Figure 50); both methods are equally acceptable. setGroundCollisionPlane(double) is the simplified method in gfSystem that allows the creation of a level plane at an input Y value; if another representation is needed (different axis), the user can create a gfCDGeomPlane at whatever position and orientation desired. Note that gfCDGeomPlanes are infinite; if a non-infinite plane is used, the creation of a thin gfCDGeomBox can be just as easily implemented, with dimensions as desired. Also note that gfCDGeomPlanes are planes that keep collision geometries on one side of them specifically, according to the input parameters. In the Figure 50 examples, collision geometries will be “on top” of the ground planes, but below the ceiling plane.

```
//a simplified method of creating a level ground plane in the Y-axis
sys->setGroundCollisionPlane(0.0);   //input parameter is Y-value of plane

 //a more generalized method of creating a ground plane, still level in the Y-axis
 //  (0x + 1y + 0z = 0)
gfCDGeom* testPlane = new gfCDGeomPlane("testPlane", 0.0f, 1.0f, 0.0f, 0.0f);
testPlane->enableCollision();

 //a ground plane that slopes 45 degrees up toward the –X-axis
 //  (1x + 1y + 0z = 0)
gfCDGeom* testPlane = new gfCDGeomPlane("testPlane", 1.0f, 1.0f, 0.0f, 0.0f);
testPlane->enableCollision();

 //a ceiling plane, which keeps all collision geometries below Y=10.0
 //  (0x + (-1)y + 0z = -10)
gfCDGeom* testPlane = new gfCDGeomPlane("testPlane", 0.0f, -1.0f, 0.0f, -10.0f);
testPlane->enableCollision();

//a method of creating a non-infinite ground plane (1000x1000 units)
gfCDGeom *boxGeom0 = new gfCDGeomBox("boxGeom0", 1000.f, 1.f, 1000.f);
boxGeom0->enableCollision();
```

Figure 50. Creating a ground plane

o) Adding Forces and Setting Positions of Physics Objects

Once physics-based objects are created, the way to move them around in the scene is to add forces to them. Forces can be added in either the World Coordinate System or the Local Coordinate System. Methods for adding and setting forces to physics-based objects are shown in Figure 51.

```
  gfObject *dynamicObject = new gfObject("dynamicObject");

//zero out all forces on a physics body
dynamicObject->zeroBody();
```

```
//set (all of) the force on a body in world coordinates
dynamicObject->setForce(fX, fY, fZ);

// set (all of) the torque on a body in world coordinates (about each axis)
dynamicObject->setTorque(fX, fY, fZ);

// add forces to a body in world coordinates
dynamicObject->addForce(fX, fY, fZ);

// add torque to a body in world coordinates (about each axis)
dynamicObject->addTorque(fX, fY, fZ);

// add forces to a body in local coordinates
dynamicObject->addRelForce(fX, fY, fZ);

// add torque to a body in local coordinates (about each axis)
dynamicObject->addRelTorque(fX, fY, fZ);

// set heading and pitch (in the physics representation)
dynamicObject->setPhysicsHeadingAndPitch(heading, pitch);
                                             //sets object to hpr of (heading, pitch, 0.0)

//set the position and orientation of the physics-based object
dynamicObject->setPhysicsPosition(posMatrix);

//get the position and orientation of the physics-based object
dynamicObject->getPhysicsPosition(posMatrix);
```

Figure 51. Methods for adding/setting forces to physics-based objects

4. Future Work Specific to the gfPhysics API

There are two basic areas in which future developments need to occur in libGF physics. The first area that needs development is in abstraction. Too much of the collision geometry and physics body functions are visible at all levels, which causes confusion as to which way to implement functionality. Currently, physics and collision can be enabled directly through gfPhysicsBody and gfCDGeom, but can also be implemented through the gfDynamic member. This can cause not only confusion but potential logic problems, as the gfDynamic member will not know when a setting is changed outside of its control.

The second area for future development is additions to functionality. Open Dynamics Engine supports the collision of triangle meshes, but the functionality to allow triangle mesh collision is not yet implemented into libGF; doing so not only involves wrapping the ODE functionality, but also includes adding the functionality to get a triangle mesh (either from a file or more directly from a scene object) and convert that mesh into usable information in a new class, gfCDGeomTrimesh. ODE also supports the application of joints, and those are not yet implemented into libGF. The structure will be an abstract base class and a set of concrete joint-type classes, in order to create all ODE supported joints, such as ball-and-socket and hinge.

F. INPUT

1. Motivation

It was decided upfront that one of the major functionalities necessary for libGF to be successful in building deployable virtual reality training applications was scalability. It would not be a robust VE system if it did not provide the ability to scale down or up as the need arose. A training VE system needs to provide the capability to run on multiple system hardware configurations, whether it be a laptop where all that is available is the standard keyboard and mouse, or it is a desktop computer using an instrumented rifle and head mounted display, both equipped with inertial trackers. This was the motivation when developing the input interface to the libGF library.

Several input interfaces were looked at when determining which would be appropriate for a deployable system. Obviously, all PCs have the use of the standard keyboard and mouse so that was the logical first interface device to account for. In addition to the keyboard and mouse, the new generation of Marines and warfighters have grown up accustomed to gaming, especially on console type systems, this made the gamepad a good choice for those accustomed the console type applications. Finally, the need to provide "realistic" training required the implementation of an interface similar to what would be used in real world room or building clearing. The obvious answer was to somehow integrate a pseudo-realistic weapon that users would feel accustomed to as an option for the third input interface device.

Providing three different input interfaces would provide VR training applications the scalability necessary to make an application deployable in any situation. Whether it is on a ship, with very limited configured on a network of laptops, or on machines configured to use gamepads, or in garrison with a fully scaled up VE instrumented rifles and head mounted displays, the interfaces could be easily interchanged, providing scalability and making the application deployable to any environment.

2. Implementation

Since the major portion of the work for this thesis was dealing with addressing the need to integrate character animation into the libGF library, the integration of the input interface devices had to be done in such a manner as to allow mapping of an inputted action to a human-like avatar action. In other words, if the user used the keyboard key or the joystick on the instrumented rifle to simulate walking forward, then the mapping of that action should be that the avatar was walking. An additional constraint was to have the ability to reconfigure the action mapping prior to, or even during, running a VE application.

Although there are several methods of handling input devices and there are different libraries available, the logical choice for handling input devices, under the Windows® operating system[19], is the Microsoft® DirectX® SDK. Because of the desire to use a library that would provide the needed functionality for both the interface input devices, as well as the networking capability, the DirectX® SDK was the library chosen for integration into libGF. It gave the ability to easily define the mapping of input devices to desired actions and the ability to reconfigure those at run-time. This provided a big benefit because now applications could be easily reconfigured for different users without the need to recompile or restart the virtual environment. Going from a right-handed person to a left-handed person would be as easy as remapping a joystick or keyboard keys.

For both the gamepad interface and the instrumented rifle interface, the Logitech® WingMan RumblePad™ [20] was chosen. In the case of instrumenting the rifle, a wireless RumplePad™ was disassembled and integrated into the rifle to provide functionality of firing the weapon and player movement through the VE using the

[19] Windows is registered to Microsoft Corporation. The term Windows is used generically for all version of the Window operating system.

[20] RumblePad is a registered trademark of Logitech, Inc. - http://www.logitech.com

joystick. Using the same gamepad for both the gamepad interface and the instrumented rifle interface, the mapping of input device to avatar actions remained consistent, and as previously mentioned, could be easily remapped through the integration of the DirectX® SDK.

Additionally, when using the fully scaled up virtual environment with the HMD, there is a need to track rifle and head movement. This tracking had to be performed in a small footprint, without any additional space constraints, this ruled out tracking systems with the requirement for overhead mounted sensors or cameras. The trackers chosen were the inertial trackers by InterSense[21]. The system used during integration was the InterSense IS-300 Pro with two inertial cubes—one for tracking rifle position and the other for head movement. The cube used for the rifle movement was mounted on the rifle barrel and provided tracking of rifle pitch and player heading. The cube for head movement was mounted to the top of the HMD and allowed the user to move their head, independent of the rifle, and provide a "look around" capability in the VE.

Figure 52. IS-300 Pro Precision Motion Tracker System (InterSense) (From Ref.)

Additional tracking systems—such as the LIBERTY™ and FASTRAK™[22] systems by Polhemus[23]—can be found that provide similar functionality and would as straightforward to integrate.

[21] InterSense, Inc. - http://www.isense.com/

[22] LIBERTY and FASTRAK are registered trademarks of Polhemus.

[23] Polhemus - http://www.polhemus.com

The keyboard, mouse and gamepad input functionality was integrated by creating a DirectX® wrapper class called diGenericClass. This new class served as the interface for the underlying DirectX® application programming interface. Encapsulating that wrapper is a generic input class, that handles device state querying for the keyboard, mouse, and gamepad, called gfInputGeneric. When a new input device is added, gfInputGeneric handles creating an instance of the DirectX® wrapper class and adding that device to the internal list of input devices. Querying of the device—to sense key presses, mouse movement, or joystick movement—is handled by calls to the DirectX® wrapper class by the gfInputGeneric interface class.

```
// Instantiating a new input device
gfInputGeneric::gfInputGeneric(const char *name)
{
        mDisplayGUI = false;
        CreateNewInput();
        if (name) SetName( name );
}

// Create a new generic input device by creating a new DirectX wrapper object
void gfInputGeneric::CreateNewInput()
{
        mNumDevices = 0;
        mGeneric = new diGenericClass();
        InputList->Add(this);
}

// Read handles querying the input device to determine the state of the device
// i.e. keypress, joystick movement, mouse movement
void gfInputGeneric::Read(const int deviceIdx,
                          DWORD *numActions,
                          DIDEVICEOBJECTDATA *deviceData)
{
        mGeneric->getState(deviceIdx, &deviceData, numActions);
}
```

Figure 53. Generic input device interface to DirectX®

The interface to the InterSense inertial trackers was accomplished by implementing the gfInputISense class which handles calls to the Isense library distributed by InterSense for use with the IS-300 Pro and inertial tracker systems. When a new tracker is added as an input device, the gfInputISense class initializes the interface for the tracker system.

```
// Creating a new Isense input device
```

```
void gfInputISense::CreateNewISense(int comm)
{

	handle = ISD_OpenTracker(NULL, comm, FALSE, FALSE);

	if(handle >0)	{
		gfNotify(GF_DEBUG, "gfInputISense created");
		mConfigured = true;
	}
	else {
		gfNotify(GF_DEBUG, "failed to init gfInputISense\n");
		mConfigured = false;
	}
}

// Add a station (inertial cube) to the input device
void gfInputISense::AddISenseStation()
{
	int success = 0;
	char command[64];

	success = ISD_SendScript(handle, command);

	if(success == 1) {
		// channel added on trackStation
		sprintf(command, "MCI%d,%d\n", trackStation, trackStation);
		success = ISD_SendScript(handle, command);

		if(success == 1) {
			// cube associated with station
			sprintf(command, "MCe\n");
			success = ISD_SendScript(handle, command);

			if(success == 1) {
				// new tracker configuration applied
				mConfigured = true;
				sprintf(command, "O%d,2,4,1\n", trackStation);
				success = ISD_SendScript(handle, command);
			}
		}
	}
	else {
		// failed to add new channel
		mConfigured = false;
	}
}
```

Figure 54. Creating a new Isense input device and adding a cube to the device

Once the tracker system is initialized and the cubes added as stations on the device, querying the device retrieves the positions of the cubes. The query returns the heading, pitch and yaw of the inertial cube in question.

```
// Read and return the position returned from querying the inertial cube
void gfInputISense::GetPosition(gfPosition *pos)
```

```
{
        if(!mConfigured) {
                // device is not configured
                return;
        }

        if(!pos) {
                // position not defined
                return;
        }

        ISD_GetData(handle, &data);

        pos->Set(0.f, 0.f, 0.f,
                mHscale * data.Station[trackStation].Orientation[0],
                mPscale * data.Station[trackStation].Orientation[1],
                mRscale * data.Station[trackStation].Orientation[2]);
}
```

Figure 55. gfInputISense handles querying the tracker for cube position

3. Application

In order to use the input interface devices in an application, they need to be implemented through the use of the gfMotionHuman motion model. The gfMotionHuman class was created to map the input actions of the different devices to the motion of a human character, so that all input from the keyboard and mouse, gamepad, or instrumented rifle is routed through that class and the appropriate actions are mapped. The ISense tracker input for the control of rifle pitch and player heading—an inertial cube mounted on the barrel of a rifle—is also captured via the gfMotionHuman class. Orientation of the ISense tracker attached to the HMD is directly input into the observer and controls the heading and pitch of where the player is looking.

a) Creating a gfMotionHuman Motion Model

To create a new gfMotionHuman:

```
// Instantiate a new gfMotionHuman motion model
gfMotionHuman *motionPtr =
        new gfMotionHuman(motionNameStr, bFlipJoystick);

// Define an input device to the motion model by using the name of
// of that input device (usually of type gfInputGeneric) to SetInput
motionPtr->SetInput(motionInputStr);

// Define a new gfPosition for the motion model
gzRefPointer<gfPosition> pos =
        new gfPosition(positionX, positionY, positionZ,
```

```
                                    positionH, positionP, positionR);

        // Set the position of the motion model
        motionPtr->Position(pos);

        // Set the initial values of the motion model
        motionPtr->SetWalkingSpeed(walkingSpeed);
        motionPtr->SetRunningSpeed(runningSpeed);
        motionPtr->SetWalkRunThreshold(walkRunThreshold);
        motionPtr->SetRotationInterval(rotationInterval);
        motionPtr->SetGlanceInterval(glanceInterval);
        motionPtr->SetSideStepInterval(sidestepInterval);
        motionPtr->SetForwardVelocity(walkingSpeed);
        motionPtr->SetRotationVelocity(rotationVelocity);
        motionPtr->SetStepUpHeight(stepUpHeight);

        // Set another input source for the motion model, in this case a Isense tracker
        motionPtr->SetInput(motionTrackerStr);
```

Figure 56. How to create a new gfMotionHuman model

Once the motion model is created, it needs to be set as the motion model for the player, or avatar character. This is accomplished by setting the motion as in Fig 51:

```
        playerPtr->SetMotion(playerMotionStr);
```

Figure 57. Setting the player motion model

Setting the tracker as input to the observer is done by passing the gfInputISense object created for that tracker into the observer, as shown in Figure 58.

```
                observerPtr->Input(input);
```

Figure 58. Setting a tracker as an input to an observer

b) The Initial Motion Model Action Mappings

Creating the motion model for the avatar and setting it as the motion model for the player automatically maps input device actions to the avatar. The initial action mappings are already defined in gfMotionHuman, but can be changed for any application and can also be changed at run-time, by running the DirectX® GUI. The DirectX® mapping set chosen was the First Person Shooter genre, with the mappings listed in the following table.

DirectX® device semantic	**Mapped to**
DIMOUSE_STEER	Heading controlled by mouse movement
DIKEYBOARD_ESCAPE	Exits application

DIBUTTON_FPS_FIRE	Fires weapon – gamepad input
DIAXIS_FPS_SIDESTEP	Side step – gamepad input
DIBUTTON_FPS_ROTATE_LEFT_LINK	Rotate left – gamepad input
DIBUTTON_FPS__ROTATE_RIGHT_LINK	Rotate right – gamepad input
DIBUTTON_FPS_FORWARD_LINK	Move forward – gamepad input
DIBUTTON_FPS_BACKWARD_LINK	Move backward – gamepad input
DIBUTTON_FPS_STEP_LEFT_LINK	Side step left – gamepad input
DIBUTTON_FPS_STEP_RIGHT_LINK	Side step right – gamepad input
DIAXIS_FPS_LOOKUPDOWN	Look up and down – gamepad input
DIAXIS_FPS_MOVE	Move – gamepad input
DIKEYBOARD_W	Move forward – keyboard
DIKEYBOARD_S	Move backward – keyboard
DIKEYBOARD_RETURN	Fire weapon – keyboard
DIKEYBOARD_LEFT	Rotate left – keyboard
DIKEYBOARD_RIGHT	Rotate right – keyboard
DIKEYBOARD_UP	Glance up – keyboard
DIKEYBOARD_DOWN	Glance down – keyboard
DIKEYBOARD_LSHIFT	Run speed vice walk speed – keyboard
DIKEYBOARD_A	Side step left – keyboard
DIKEYBOARD_D	Side step right – keyboard
DIAXIS_FPS_ROTATE	Rotate – gamepad input
DIMOUSE_YAXIS	Glance up and down – mouse
DIMOUSE_BUTTON0	Fire weapon – mouse
DIKEYBOARD_F1	Bring up DirectX® GUI - keyboard

Table 2. Initial input device mappings for DirectX®

These mappings are initialized in gfMotionHuman by creating an array of actions that DirectX® can use to connect input device actions to character actions. Each array entry contains the action enumeration, string representing the action, and the action mapping semantic (as defined by DirectX®).

c) Defining the Mappings

As the motion model receives inputs from the interface devices attached, it checks the list of available action mappings and, if a match is found, it performs the steps defined for that action mapping. An example of how to map actions to input devices is shown in Figure 59.

```
// Defines the mapping for the keyboard 'D' key to be mapped to side step right
mActionMapping[46].uAppData = eB_SIDESTEP_RIGHT;
mActionMapping[46].dwSemantic = DIKEYBOARD_D;
mActionMapping[46].lptszActionName = "StepRightLink";

// Defines the mapping for the gamepad rotate action to be mapped to rotate
mActionMapping[47].uAppData = eA_ROTATE;
mActionMapping[47].dwSemantic = DIAXIS_FPS_ROTATE;
mActionMapping[47].lptszActionName = "Rotate";
```

Figure 59. Example action mapping in gfMotionHuman

d) Handling Actions for Input Devices

The actions performed for each mapping can vary from setting the position of the motion model, and thereby the player, to sending a message to start a character animation sequence. Any other object set that has subscribed to the motion model messages will be able to receive, and act upon, the SendNotify messages. This is how actions are mapped from input device to motion model to character animation or to the network. During the Update() method of the gfMotionHuman class, each device is queried to determine its input to the motion model for that current frame.

When the ISense tracker is attached, and initialized to provide input to the model motion as described above, the position is queried and used to set the heading and pitch of the rifle. Specifically, it first finds the gfInputISense object used for the rifle, gets the current orientation of the inertial cube, passes the heading and pitch to the physics engine, and then resets the rifle offset. In addition, a notification message is also sent to notify any objects subscribed to be notified that the rifle orientation has changed.

```
// Get the gfInputISense defined for the tracker
gfInputISense *rifle = (gfInputISense*)mInput->Get(i);

// Get the current position of the rifle
rifle->GetPosition(riflePos);

// Create a new gfPosition and set the position with the new heading
gzRefPointer<gfPosition> currentPos = new gfPosition();
currentPos->Set(mPosition->X(), mPosition->Y(), mPosition->Z(),
                riflePos->H(),  mPosition->P(), mPosition->R());

// Pass new heading and pitch to physics engine
setPhysicsHeadingAndPitch(riflePos->H(),  mPosition->P());

// Set the new rifle offset
mRifleOffset->Set(mRifleOffset->X(), mRifleOffset->Y(), mRifleOffset->Z(),
                  mRifleOffset->H(), riflePos->P(),     mRifleOffset->R());

// Send message notification of new rifle position
SendNotify("rifleaim", riflePos);
```

Figure 60. Resetting the rifle offset based on the inertial cube

Actions for devices covered by the gfInputGeneric class are handled differently. The action returned from the device is mapped according to the mappings setup via the action mapping discussed earlier. When a device action is received, the applicable enumerated action name is used to define the action to be taken. Actions can be easily changed or added by modifying the action mapping array and then adding the enumerated action name to the switch statement, as shown below.

```
// Switch on enumerated actions returned by each device
switch (deviceAction) {
// Steering from the mouse, so set the relative X value to later update position
case eA_STEER:
        relX = (int)deviceData[action].dwData;
        break;

// Glance up/down received from mouse, send out notification
case eA_GLANCEUPDOWN:
        mousePitch = (int)deviceData[action].dwData;

        data->SetAction(eA_GLANCEUPDOWN);
        SendNotify("GlanceUpDown", data);
        break;

// Rotate from mouse received,  set new absolute X for updating position and
// send out notification
case eA_ROTATE:
        absX = (int)deviceData[action].dwData;

        if(absX < MIN_INPUT_THRESHOLD && absX > -MIN_INPUT_THRESHOLD) {
                absX = 0;
```

```
        }

        data->SetAction(eA_ROTATE);
        SendNotify("Joystick - Rotate", data);
        break;
}
```

Figure 61. Handling received actions from gfInputGeneric devices

e) A Motion Model other than gfMotionHuman

Although the mappings for gfMotionHuman have been defined to emulate the actions that would be taken by a human character, by modifying the action mappings and the actions to perform, the motion model can easily be modified to emulate a helicopter or HMMWV. An example would be—instead of mapping the left and right movement of the mouse to the rotation of a character, by changing the mapping and the actions, it could easily be used to steer a HMMWV.

4. Future Work

Work left to be done in the input portion libGF lies mostly with implementing wireless functionality. Currently, the trackers and joystick gamepad are tethered by the wires that connect them to the computer. These wires can pose a problem with entangling the user when making several consecutive turns in the virtual environment, if an additional person is not close by to ensure that the wires are periodically untangled. Switching to a completely wireless implementation would eliminate this problem and would greatly increase the free movement inside the virtual environment and also the realism of the immersion. Having to stop a building clearing exercise to untangle wires is a quick reminder that it is only a simulated environment and not a real, or fully immersive, exercise.

III. SYSTEM USABILITY ANALYSIS

A. INTRODUCTION

An experiment was conducted on a group of eight subjects to determine the maneuverability in a virtual tactical environment using different hardware interfaces. Users were asked to complete a series of tasks to demonstrate their ability to maneuver and perform building clearing procedures. The interest of this experiment was to determine which of the different hardware interfaces proved to provide the best maneuverability through the virtual MOUT environment. The experiment did not address the subjects ability to tactically clear the building as would be expected in a building clearing exercise, only the differences in a user's ability to maneuver based on the hardware interface used. For this experiment, three different interfaces were used: standard keyboard and mouse; GamePad; instrumented (joystick enabled) rifle with Head-Mounted Display (HMD) using InterSense trackers.

B. BACKGROUND

1. Subjects

Subjects chosen for this experiment were Marine Corps Officers attending the Naval Postgraduate School. The reasoning behind choosing Marines was their previous exposure to MOUT situations, which allowed for the tactical movement desired through the environment without additional training. Although we wanted the participants to move as tactically as possible through the environment, we made no effort to measure this, and, since it was of low relevance to maneuverability, we trusted and were satisfied with the fact that the similar training of all participants gave us the similarity of tactical movement that we were looking for in participants.

2. Hardware

The computer used for this experiment was a small footprint Shuttle XPC configured with an AMD Athlon XP 2500 Barton 333MHz PSB Processor, 512 MB

RAM, 60 Gb hard drive and an XFX Nvidia FX 5600 graphics card (128 MB version). The operating system was Microsoft Windows XP. The monitor used during the experiment was a Dell Active matrix LCD display, Model 2000FP. Measurements for the display are 16.1 inches horizontal, 12.1 inches vertical, and 20.1 inches diagonal. The users were approximately 20 inches from the display, giving an apparent Field of View (FOV) of 34 degress. View frustrum set in the environment is 33 degress.

Standard keyboard and optical mouse were used for the first hardware interface in this experiment. They were configured so that the 'W', 'S', 'A', and 'D' keys provided forward, reverse, left and right movement respectfully. Holding down the 'Shift' key caused the subject to move in a tactical run speed vice the normal tactical walking speed. Moving the mouse changed the orientation (heading and pitch) of the avatar in the environment and allowed for looking around and changing the pitch of the rifle when aiming. The left mouse button fired the weapon.

The second interface used was a typical gaming interface – a gamepad. The version used in this experiment was the Logitech Wingman Rumblepad. The configuration of the gamepad was the left joystick controlled the forward, reverse, and rotation movement. The right joystick changed the pitch of the rifle – for aiming. Firing was accomplished via the 'A' button.

The third interface was the instrumented rifle with HMD and trackers. The rifle was an Airsoft rifle outfitted with the joystick and button controls from a Wireless Wingman Rumblepad. The firing of the rifle was connected the trigger of the rifle. A joystick was mounted the end of the rifle handgrip allowing the subject to control movement via a thumb-controlled joystick, with provision to support both right and left-handed operators. The joystick was mapped to control the forward, reverse, left and right movement in the environment. The HMD used was a 5DT Head Mounted Display Model DH-4400VPD. The resolution on this HMD was 800 by 600, and the apparent FOV was 32 degrees, diagonally, with a 4:3 aspect ratio. Trackers were used in conjunction with the joystick to control the movement of the rifle and the subject's look-at direction in the environment. The system used for the experiment was an InterSense IS-300 Pro Inertial Tracking System with two inertial cubes; one cube was placed on the top of the rifle barrel to control the orientation of the rifle and one cube was placed on the top of the HMD to control the look-at direction of the avatar. Both inertial cubes were initialized and bore-sighted prior to each subject conducting the experiment.

3. Environment

The virtual environment used for this experiment was one developed using the libGF open-source graphics library developed at the Naval Postgraduate School. The environment model was a five building MOUT site created for this experiment (see Figure 62). Only the buildings labeled 1, 2, and 3 were used for this experiment.

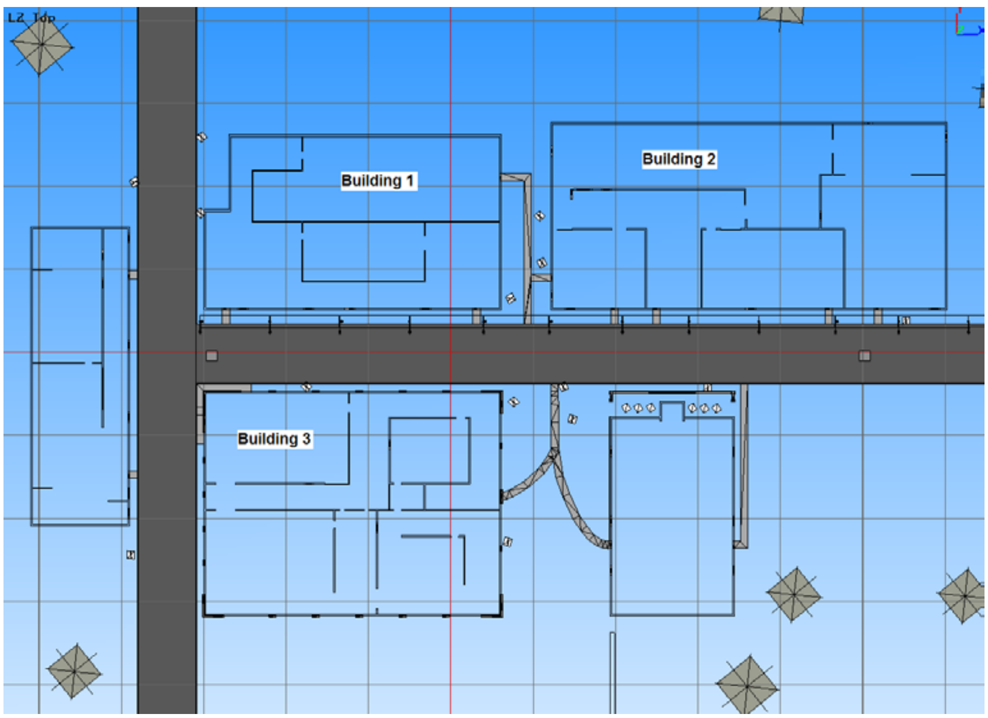


Figure 62. MOUT Overhead

Targets were placed throughout the three buildings in locations that required good tactical movement. Specific locations of the targets can been seen in the individual briefing sheets for each segment. Targets in the environment had the appearance of simple plywood targets placed on the posts, which fell when shot (see Figure 63).

Figure 63. Target

The display resolution was set to 800x600 for each segment of the experiment because of the limitations of the HMD and to provide consistency throughout the experiment.

C. EXPERIMENT

1. In Briefing

Prior to conducting the experiment, each subject was given an in briefing explaining the purpose of the experiment and what they would be asked to accomplish during the experiment. Each subject was briefed on the minimal risk associated with wearing an HMD for an extended period of time, which for this experiment was expected to be 20 minutes for each subject.

Each subject was asked to complete a preliminary questionnaire used for background information to determine their experience with computer gaming, experience with actual training in MOUT, and pervious exposure to virtual environments and head mounted displays.

2. Completing the Tasks

Subjects were given a series of three segments to accomplish per hardware interface. The order of the interface use was chosen in random prior to the start of the experiment. Each interface was associated with a different building, which remained constant for each subject. The three segments conducted for each interface were conducted twice, sequentially, to determine if building familiarity played any role in the subjects ease in maneuvering through the building and conducted each segment.

For each segment, the time to complete the task was noted, along with the subject's rating of the segment's ease compared previous segments while using the same interface. After the run of each set of segments, using the same interface, the subject's were asked to complete a second run of the same set of segments.

3. User Questionnaire

Upon completion of all tasks, the subjects were asked to fill out a questionnaire. The questionnaires asked the subjects to rate the after effects of wearing the HDM, the realism of the movement and field of view inside the environment for each of the three different hardware interfaces. They were also asked to rate the ease of accomplishing each task, maneuvering through doorways and around obstructions, ease of tactical movement, and ability to sight in and engage targets.

D. RESULTS

1. Subject Profile

There were eight subjects that were used to perform this experiment. Of those:

- 100% of the subjects were male.
- 100% of the subjects were U.S. Marine Corps officers.
- 100% of the subjects had more than 50 hours of experience in a MOUT environment, with two subjects having more than 150 hours.
- All but one subject had been previously exposed to virtual environments, but none had been exposed to a virtual environment using a head-mounted display.
- 100% of the subjects spent less than two hours per month playing first person shooter type games.
- Only three of the eight subjects averaged more than 2 to 4 hours of computer usage per day.

2. Subject Questionnaire Results

The following conclusions were drawn from the post experiment questionnaires filled out by each subject:

- Movement rate of the HMD and rifle was more realistic than the keyboard/mouse and gamepad.
- A more immersive feeling was obtained while using the HMD and rifle.
- The movement using the HDM and rifle was more realistic in the VE than using the other two interfaces.
- Subjects found the tasks pretty comparable to accomplish for both the keyboard/mouse and the HMD and rifle.
- Maneuvering through doors was found to be the easiest using the HDM and rifle.
- The difficulty of maneuvering around obstacles was found to be the same for both the HMD and rifle and the keyboard/mouse.

- Tactical movement was best with the HMD and rifle (but only slightly easier than keyboard/mouse).
- Sighting and engaging the targets was found to be easier with the HMD and rifle.

3. Statistical Results

Significant statistical analysis is achieved by Oneway Anova analysis of the data. Analysis was done on the three separate buildings; Building 1 was representative of the keyboard/mouse input setup; Building 2 was representative of the gamepad setup; Building 3 was representative of the HMD and instrumented rifle setup. Data collected represents the individual's position at each second of the experiment. Intent was to collect information on position, time, and discontinuities. Discontinuities can be described as variance in heading above a given threshold. For statistical analysis, a lower threshold of 45 degrees and a higher threshold of 90 degrees were used.

The nature of tactical maneuver and, specifically, building clearing tasks is discontinuous by nature. The need to "pie off" an entrance or opening requires the participant to constantly change heading while making small positional changes. Many other such examples of discontinuous movement can be found in the conduct of tactical maneuver and building clearing tasks. The need for such discontinuous maneuverability means that the better input device choice will be most likely found in the choice that allows for more rapid discontinuity over time.

The expectation is that a lower total time would be representative of the more maneuverable input device setup. Since however, this is a discontinuous task, discontinuities per second reveals the immediacy of maneuverability across the three input setups. Further, a higher discontinuity per second rate is only desirable if the total time is similar across the experiment. The following graphs display the analysis of the experiment.

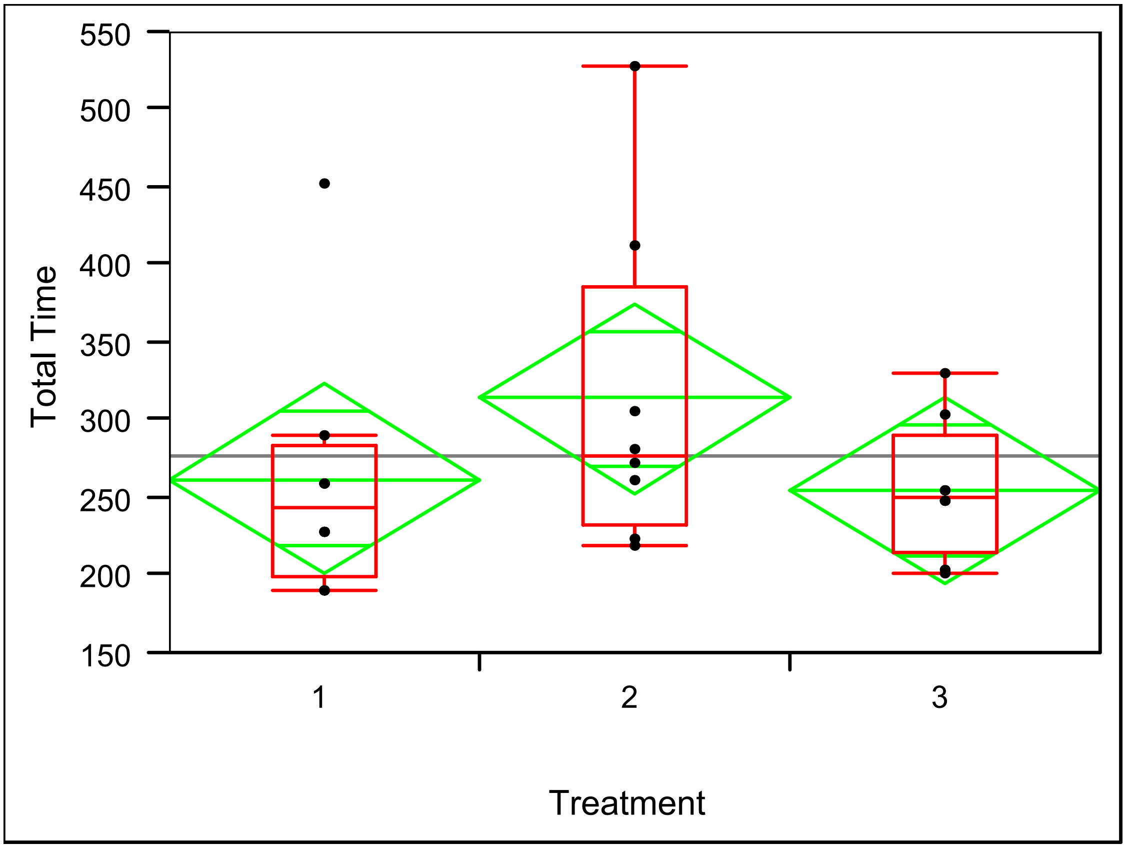


Figure 64. Oneway Analysis of Total Time By Treatment at 90 degrees

An analysis of variance of total time by treatment results in P = 0.3191 ($F(2,23) = 1.2066$) suggesting that there is no significant difference between treatments in terms of total time.

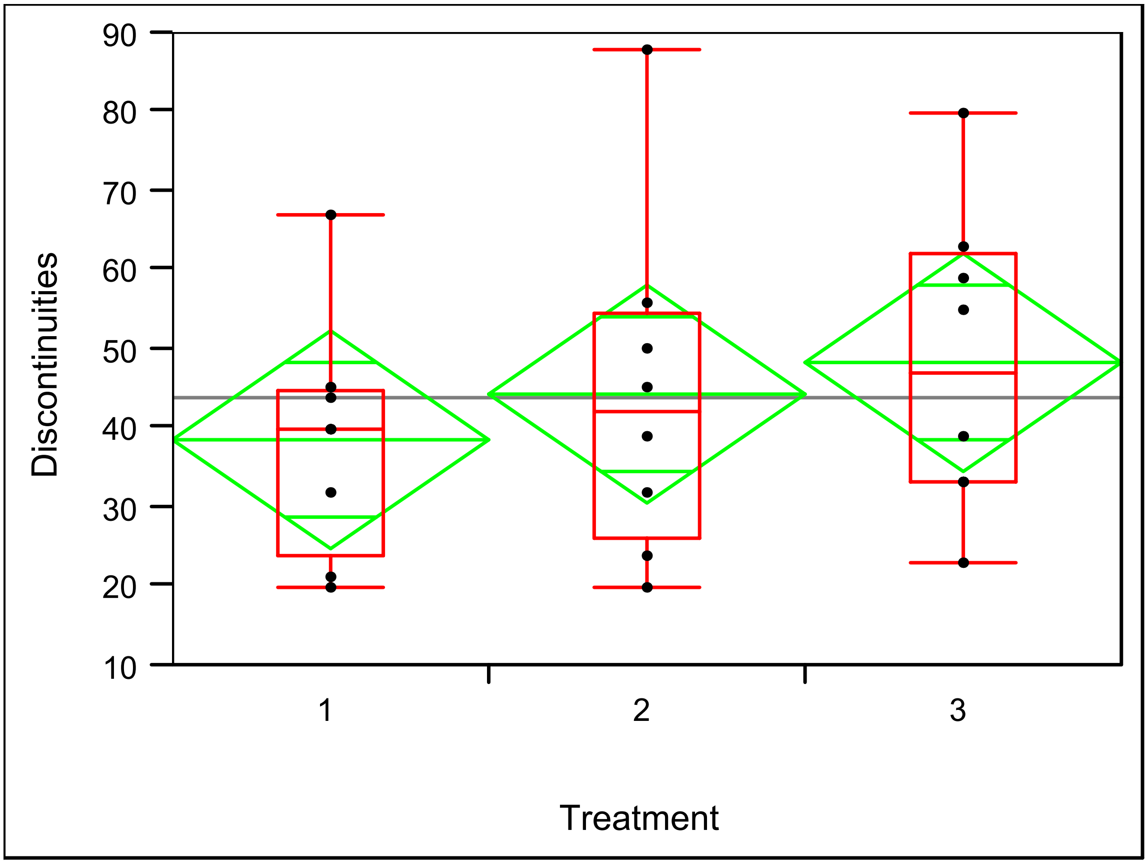


Figure 65. Oneway Analysis of Discontinuities By Treatment at 90 degrees

An analysis of variance of discontinuities by treatment results in P = 0.6038 (F(2,23) = 0.5169) suggesting that there is no significant difference between treatments in terms of total discontinuities.

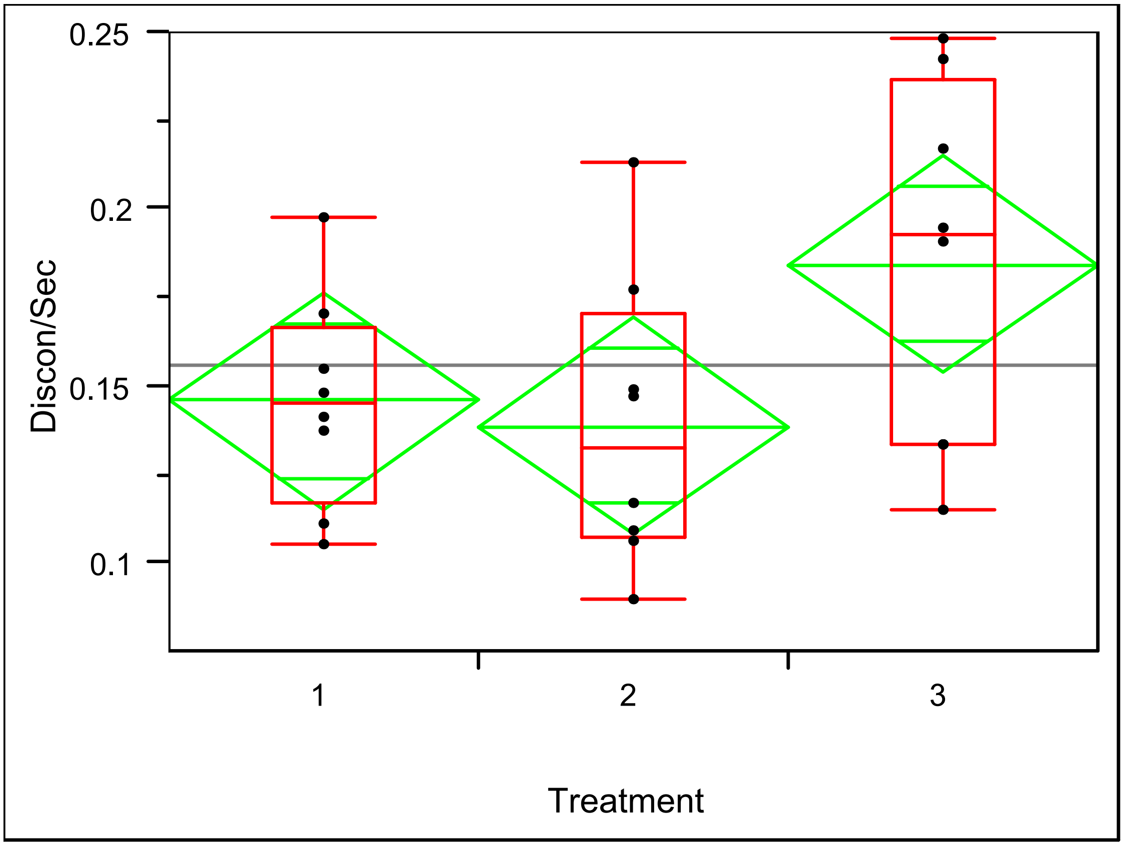


Figure 66. Oneway Analysis of Discontinuities/Sec By Treatment at 90 degrees

An analysis of discontinuities per second by treatment results in $P \leq 0.1$ ($F(2,23) = 2.7645$) showing that treatment 3 (HMD and instrumented rifle) had the greatest discontinuities per second; this is significant because the analysis of time showed no significant increase of the same treatment.

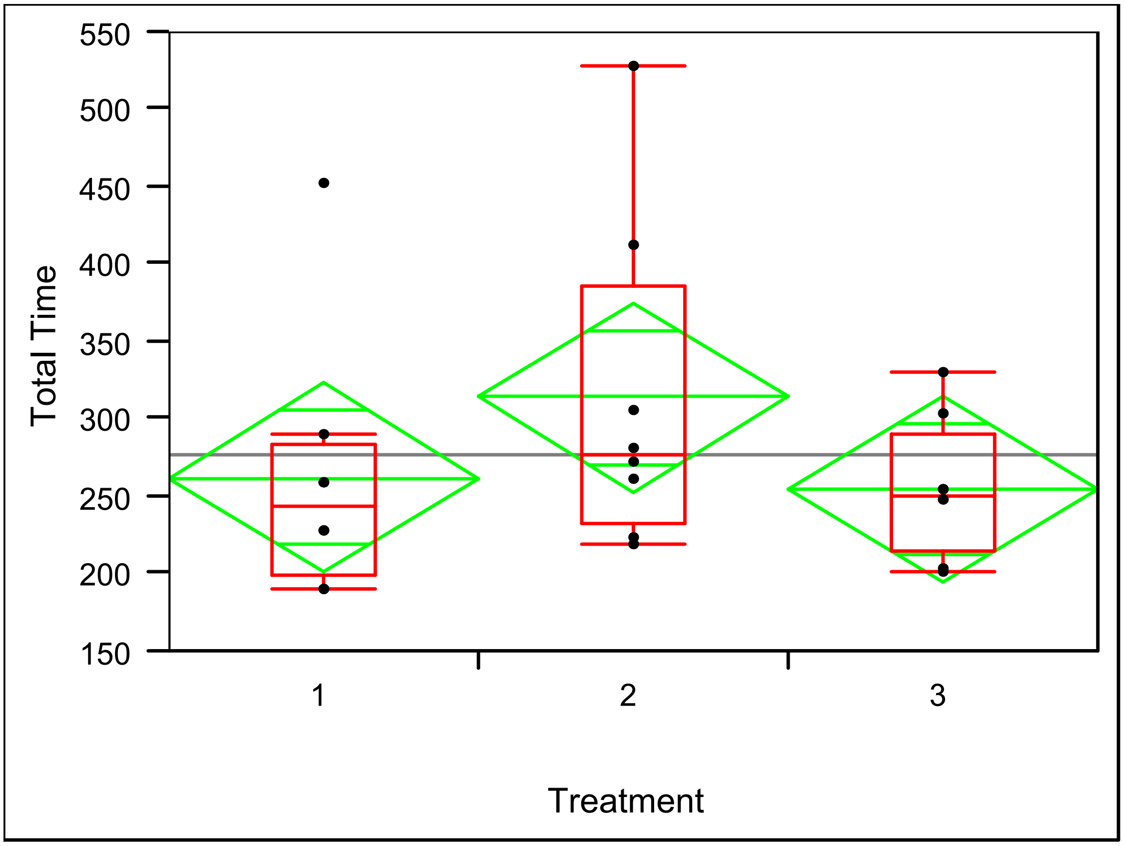


Figure 67. Oneway Analysis of Total Time By Treatment at 45 degrees

Repeating the same statistical tests on the 45 degrees measurement, an analysis of variance of total time by treatment results in P = 0.3191 (F(2,23) = 1.2066) suggesting that there is no significant difference between treatments. This data matches that of the plot for 90 degrees because the total time does not change with variance of the discontinuity threshold.

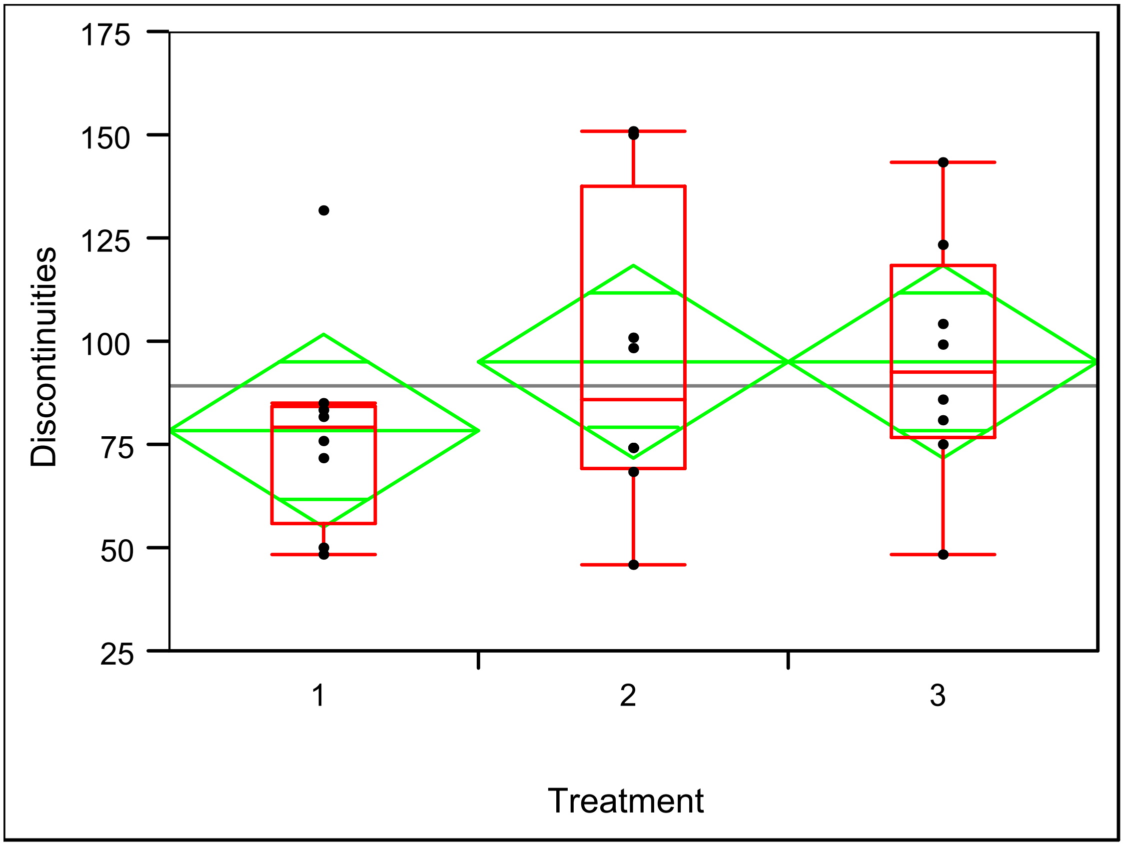


Figure 68. Oneway Analysis of Discontinuities By Treatment at 45 degrees

An analysis of variance of discontinuities by treatment results in P = 0.4932 (F(2,23) = 0.7312) suggesting that there is no significant difference between treatments. This analysis shows no significant statistical difference from the 90 degree threshold.

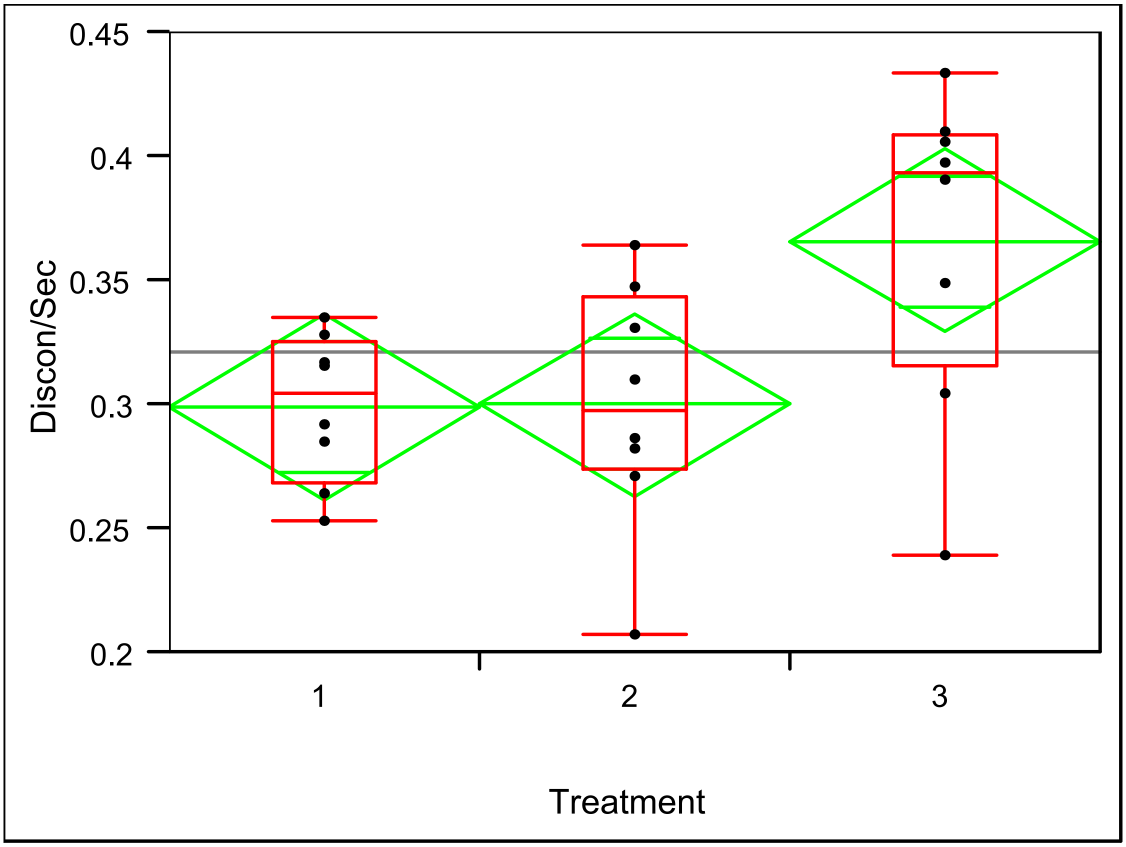


Figure 69. Oneway Analysis of Discontinuities/Sec By Treatment at 45 degrees

An analysis of discontinuities per second by treatment results in $P \leq 0.1$ ($F(2,23) = 4.6492$) showing that treatment 3 had the greatest discontinuities per second; this is significant, as with 90 degrees, because the analysis of time showed no significant increase of the same treatment. The fact that the two end thresholds produce the same statistical analysis shows that there is no significance in what threshold is used.

Given that building clearing is a necessarily discontinuous task, statistical analysis shows that the HMD display and instrumented rifle setup proves to be the more realistic interface for use in this task. It is important to note that this does not validate the ability of this application as a trainer for the live equivalent of this task – merely that the HMD and rifle setup is closer to real world task execution in this application.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. CONCLUSIONS AND RECOMMENDATIONS

A. MILITARY TRAINING COMMANDS HAVE VIABLE VIRTUAL REALITY ALTERNATIVES

The military is no longer locked into either real training or a two-dimensional alternative. Too often, when a unit does not have the ability to train in a real environment, the alternative is insufficient to maintain proficiency. The tools are now available to overcome these deficiencies due to prolonged absence from real training environments. Virtual reality tools are capable of providing a realistic environment in which participants can feel present and can learn skills that they could otherwise learn in a real environment.

There is no premise to say that virtual training will ever replace real training; that is not the issue. The issue is that real training is becoming more difficult to conduct due to schedules, money, and resources. There are many times when units would train if given the opportunity, but the opportunity does not exist. This is a twofold problem; training in those conditions leads to atrophy of skills between training evolutions, which further serves to reduce the effectiveness of training that is accomplished because units cannot simply move forward and get more out of each training session when fundamentals need to be relearned. However, if skills could be maintained and basic procedural skills could even be added to between training evolutions, real training evolutions would have a much stronger impact on unit capability and readiness.

Virtual reality training environments serve this purpose; they are capable of preventing atrophy of skills and of covering new skills in advance of real training scenarios. Virtual tools can be modified to fit a particular need; they can be scaled up to larger environments, or scaled down for single participants. They can also be scaled to work on different platforms and with varying equipment based on need and availability. Virtual reality training environment provide a versatile way for units to maintain readiness skills. Since there is a diversity of products used to make these virtual environments, there are alternatives to choose from based on need and cost.

B. VIRTUAL REALITY TRAINING TOOLS DO NOT HAVE TO BE EXPENSIVE

When virtual reality was a fairly new term to a user market, proprietary software was the only way to ensure that the user was getting what he expected. Different companies provided different results, and buyers paid for the level of capability that they wanted. Commercial vendors could write whatever interface they liked and could provide whatever functionality they liked so long as it sold. This created a lack of flexibility for the buyer; once a buyer was locked into a particular brand of software, the buyer could only work with the functionalities and capabilities provided by the software vendor, unless the buyer wanted to pay more to have additional functionality written.

This lack of flexibility and the commercial costs that commercial software vendors could charge because of it led to the creation of a market of simulation engines trying to make a better product. Eventually, though, because alternatives became available, competition turned out to be the cause for commoditization, with each software company trying to provide the same underlying virtual reality capabilities as all others.

With open-source simulations engines being written that—because of commoditization—provide the same functionality as expensive commercial software, the only difference is the level of service from the software vendor. So the only real cost that now matters is the cost associated with a level of service adequate to ensure that software is up-to-date, stable, and capable. Because many open-source alternatives have members who consult professionally, the same level of service and customer commitment can be met by open-source alternatives as by commercial vendors, but the cost can be significantly lower. The DoD pays for either its own management and maintenance of the software or for consultant support, both of which can be lower cost solutions to commercial software.

V. FUTURE WORK

A. REORGANIZATION OF THE ARCHITECTURE

libGF has evolved into an architecture that is fully capable of creating virtual environments. What has been described in this work is, for the most part, work that the authors conducted in adding to the libGF architecture (with exception to much of the basic API that is discussed in the Quick-Start User Manual). However, this is only a portion of the work that has been contributed to libGF. Because a number of people have contributed work to the libGF architecture, the organization of the libGF structure lends itself to the need to redesign the architecture from the bottom up, with emphasis on what features to make available and where those features fit into a new structure.

Currently, a new simulation engine is being designed and written, with libGF being used as background for how to engineer a new system. Sections of libGF that were well architectured and have a structure that lend themselves to a new system are being ported or rewritten. Sections that were seen as needing additional work or needing rewritten in libGF are being analyzed for significance as to what was originally done correctly or incorrectly, and are then being discarded or rewritten in the new system. New functionality that was discussed but never reached in libGF is being discussed early in the organizational cycle of the new system, so that sections are not overlooked.

What will hopefully come out of the successor to libGF is a production system capable of being used to create virtual training environments. This was the goal of libGF, and is the continuing goal of follow-on work. A new system written to reach a level of production could be handed to DoD software organizations, such as simulation software engineering labs, so that those organizations have the ability to create new virtual reality training environments from which to conduct further study or from which service personnel can be trained. The ability of those organizations to create virtual reality training environments at a low-cost equates to the addition of low-cost virtual training capability at the unit level, where it can be used to maintain procedural skills, enhance training levels, and multiply the benefits of live training.

B. DETERMINING WHETHER APPLICATIONS BUILT ON LIBGF PROVIDE POSITIVE TRAINING TRANSFER

The assumption made, when discussing the benefits received through virtual training, is the positive benefit in unit and individual capability, which can be further beneficial to follow-on live training. This is not, however, a proven statement. Little study has, in fact, been conducted and documented on the effects of virtual training and the impact of that training on unit ability. The Marine Corps has been using the Indoor Simulated Marksmanship Trainer (ISMT) for years as a practical addition to marksmanship training; recently, in some units, this system has been so enveloped into the training program that practice on the ISMT is a requirement of some Marines prior to live-fire marksmanship training. The assumption is clearly made that the addition of virtual environment training is beneficial to live-fire training, but there has not been actual, quantifiable study conducted on how beneficial that virtual training really is.

This is not a terribly surprising revelation; an understanding of the use of software and computer systems has eluded U.S. military organizations until very recently. While software and hardware have advanced dramatically in recent years, the military has been slow to realize that fact or its significance. The military acquisition process is a classic example; tracking software development as a significant part of acquisition programs has only become a serious issue very recently. Until that time, software and its integration were seen as a small and fairly insignificant portion of the acquisition process. Nothing could be further from the truth, and the same can be said of the use of software products such as virtual training environments in training.

While the authors believe that what they have accomplished can be beneficial to individual and unit readiness, this has yet to be proven. Study is now possible, however, with applications built using the libGF simulation engine. By using geometry modeled on the buildings and scenery of an actual training environment, future studies can be conducted on virtual environment training in a replica environment, and whether that training has impact on individual or unit capability when subsequently training in the live environment. The findings of such studies could have significant impact on future military training and the use of virtual training environments.

LIST OF REFERENCES

Ascension Technology Corp. (no date). Ascension Technology (online). http://www.ascension-tech.com (2002, Sep. 5).

Anderson, Paul. Visualization; Virtual Reality Training for the Future. ENN Daily Report, Vol 2, No. 306. http://www.emergency.com/vr-train.htm. (1996, Nov. 1).

Aukstakalnis, S., and Blatner, D. (1992), Silicon Mirage: The Art and Science of Virtual Reality, Berkeley: Peachpit Press.

Bandi, Srikanth. (2000). Path Finding for Human Motion in Virtual Environments. Computaional Geometry 15, 103-127.

Baumann, Jim, Military applications of virtual reality, Human Interface Technology Laboratory. http://www.hitl.washington.edu/scivw/EVE/II.G.Military.html. (Fall 1993).

Bell, H. H., Mastaglio, T. W., and Moses, F. (1993), "Using distributed interactive simulations for joint service training", Military Simulation & Training, 5, 28-30.

Comet, Michael B. (1999). Character Animation: Principles and Practice (online). http://www.comet-cartoons.com/toons/3ddocs/charanim (2002, Aug. 27).

Crandol, Michael. (1999). The History of Animation: Advantages and Disadvantages of th eStudio System in the Production of an Art Form, Digital Media FX – The Power of Imagination (online). http://www.digitalmediafx.com/Features/animationhistoryp.html> (2002, Aug. 27).

Creating Grace: Baginski Animates The Cathedral. (Jul. 2002). Animation Magazine, 29.

Deployable Virtual Training Environment. Coalescent Technologies Corporation. http://www.ctcorp.com/performance15.html. (Modified 10 Jun. 2003).

Dollner, Jurgen, and Hinrichs, Klaus. A Generalized Scene Graph. Institut fur Informatik, Unversitat Munster.

Eyetronics (no date). Eyetronics: 3D Scanning Solutions (online). <www.eyetronics.com> (2002, Sep. 5).

Fisher, J. Brian. "Using Virtual Reality to Train Air Traffic Controllers." http://www.tss.swri.edu/pub/2001iats_atcvr.htm. (2001).

Gleicher, Michael. (no date). Animation From Observation: Motion Capture and Motion Editing (online). www.awn.com/mag/issue3.11/3.11pages/kenyonrosehtal.php3> (2002, Aug. 29).

Henry-Biskup, Stefan. (Nov. 1998). Anatomically Correct Character Modeling (online). http://www.gamasutra.com/features/visual_arts/19981113/charmod_01.htm Gamasutra Vol 2, Issue 45.

Hogue, Jeff. Parachute Flight Training Simulation. Systems Technology, Inc. http://www.systemstech.com/paramain.htm. (2003, Apr. 29).

James, Patrick. (1997). History of Animation: Before Disney (online). http://www-viz.tamu.edu/courses/viza615/97spring/pjames/history/main.html (2002, Aug. 27).

Johnson, David M. (1997, Feb.), Learning in a synthetic environment : the effect of visual display, presence, and simulator sickness, Corporate author: U.S. Army Research Institute for the Behavioral and Social Sciences. Rotary-Wing Aviation Research Unit, (Technical report {U.S. Army Research Institute for the Behavioral and Social Sciences} ; 1057), (Army project number 2O262785A791), Alexandria, Va. : U.S. Army Research Institute for the Behavioral and Social Sciences, 1997.

Knerr, Bruce W. …[et al.] (1998, Nov.), Virtual environments for dismounted soldier training and performance : results, recommendations, and issues, Corporate author: U.S. Army Research Institute for the Behavioral and Social Sciences, (Technical report {U.S. Army Research Institute for the Behavioral and Social Sciences} ; 1089), Alexandria, Va. : U.S. Army Research Institute for the Behavioral and Social Sciences, 1998.

Lampton, Donald R. …[et al.] (2001, Mar.), Instructional Strategies for Training Teams in Virtual Environments, Corporate author: U.S. Army Research Institute for the Behavioral and Social Sciences (Technical report {U.S. Army Research Institute for the Behavioral and Social Sciences} ; 1110), Alexandria, Va. : U.S. Army Research Institute for the Behavioral and Social Sciences, [2001].

Landauer, Christopers, Bellman Kirstie L. (Eds) (1998), Virtual Worlds and Simulation Conference (VWSIM '98), (Full title: Proceedings of the Virtual Worlds and Simulation Conference (VWSIM '98) ; 1998 Western Multiconference, San Diego, California, January 11-14, 1998, Catamaran Resort Hotel), Simulation series ; v. 30, no. 2, San Diego, Calif. : The Society for Computer Simulation International, 1998.

Landauer, Christopers, Bellman Kirstie L. (Eds) (1999), Virtual Worlds and Simulation Conference (VWSIM '99), (Full title: Proceedings of the Virtual Worlds and

Simulation Conference (VWSIM '99) ; 1999 Western Multiconference, San Francisco, California, January 17-20, 1999, Cathedral Hill Hotel), Simulation series ; v. 31, no. 2, San Diego, Calif. : The Society for Computer Simulation International, 1999.

Lind, Judith H. (1995, Sept.), Battlefield behavior of neutrals and hostiles : models for the team tactical engagement simulator (TTES), Corporate Author: Naval Postgraduate School (U.S.) (NPS-OR-95-006), Monterey, Calif. : Naval Postgraduate School, [1995].

Norris, Steven D. (1998, Sept.), A task analysis of underway replenishment for virtual environment ship-handling simulator scenario development, Thesis (M.S. in Computer Science) Naval Postgraduate School, Monterey, Calif. : Naval Postgraduate School ; Springfield, Va. : Available from National Technical Information Service, 1998, http://library.nps.navy.mil/uhtbin/hyperion-image/98Sep_Norris.pdf (5.74 MB) or http://handle.dtic.mil/100.2/ADA355905.

Piau, Pang Sie. (2001, Oct.). Hollywood Industry: Introduction to Modeling 3D Characters (online). http://www.hollywoodindustry.com/ (2002, Sep. 5).

Pleban, Robert J. (2001, Mar.), Training and Assessment of Decision-Making Skills in Virtual Environments, (Army Project Number 2O262785A790), Alexandria, Va. : U.S. Army Research Institute for the Behavioral and Social Sciences, http://handle.dtic.mil/100.2/ADA389677.

Ripley, Tim. (2003, Spring). Creating the Virtual Battlefield. http://www.ets-news.com/virtual.htm. (2003, Sep. 16).

Sanders, Richard D. Jr. and Scorgie, Mark A. (2002, Mar.), The effect of sound delivery methods on a user's sense of presence in a virtual environment, Thesis (M.S. in Modeling, Virtual Environments, and Simulation)--Naval Postgraduate School, Monterey, Calif. : Naval Postgraduate School ; Springfield, Va. : Available from National Technical Information Service, 2002, http://library.nps.navy.mil/uhtbin/hyperion-image/02Mar_Sanders.pdf(1.18 MB).

Scott, Joanna. (2002, Jul.). “Spider-Man: The Inside Story.” 3D World, 18-21.

“Shiphandling and Advanced Shiphandling (SH/ASH).” Navy Course Descriptions. Marine Safety International. http://www.marinesafety.com/sections/usn/usn_SH.htm.

Shlechter, Theodore M. …[et al.] (1997, Sept.), An examination of training issues associated with the virtual training program, Corporate author: U.S. Army Research Institute for the Behavioral and Social Sciences, (Technical report {U.S. Army Research Institute for the Behavioral and Social Sciences} ; 1072), Alexandria, Va. : U.S. Army Research Institute for the Behavioral and Social Sciences, 1997.

Singhal, Sandeep and Zyda, Michael. Networked Virtual Environments - Design and Implementation. New York: ACM Press Books, SIGGRAPH Series. 23 July 1999. ISBN 0-201-32557-8. 315 pages.

Street, Rita. (2002, Jul.). Not So Far Away: Animating the Future at ILM. Animation Magazine, 40-44.

Stuman, David J. (1994). A Brief History of Motion Capture for Computer Character Animation. Character Motion Systems, SIGGRAPH 94: Course 9, http://www.siggraph.org/education/materials/HyperGraph/animation/character_animation/motion_capture/history1.htm (2002, Aug. 27).

Sullivan, CDR Joseph A (USN). Compendium. "Helicopter Virtual Environment and Navigation Studies at NPS." http://www.nps.navy.mil/cs/sullivan/HeloNav.html.

"Tactical 3D Simulation Environment—VBS1™." Coalescent Technologies Corporation. http://www.ctcorp.com/capability20.html. (Modified 10 Jun. 2003).

"Tank Driver Trainers (M1 and M1A2 TDT)." PM CATT, Project Manager Combined Arms Tactical Trainer. http://www.stricom.army.mil/PRODUCTS/TDT/. (Modified 8 Jul. 2003).

Tu, Xiaoyuan. (1999). Artificial Animals for Computer Animation: Biomechanics, Locomotion, Perception, and Behavior. In G. Goss, J. Hartmanis and J. van Leeuwen (Eds.), Lecture Notes in Computer Science 1635. New York: Springer.

U.S. Congress, Office of Technology Assessment, Virtual Reality and Technologies for Combat Simulation--Background Paper, OTA-BP-ISS-136 (Washington, DC: U.S. Government Printing Office, Sept. 1994), http://www.wws.princeton.edu/cgi-bin/byteserv.prl/~ota/disk1/1994/9444/9444.PDF.

"U.S. Navy SURFLANT Norfolk, VA." Marine Safety International. http://www.marinesafety.com/sections/usn/usn_norfolk.html.

Washington, David B. (2001, Sept.), Implementation of a multi-agent simulation for the NPSNET-V virtual environment research project, Thesis (M.S. in Computer Science) Naval Postgraduate School, Monterey, Calif. : Naval Postgraduate School ; Springfield, Va. : Available from National Technical Information Service, 2001, http://theses.nps.navy.mil/Thesis_01sep_Washington.pdf(835 KB).

APPENDIX A. LIBGF QUICK-START USER MANUAL

A. REQUIRED SETUP FOR DEVELOPMENT

1. Setting up WinCVS to Download the Source Code

The libGF software resides at http://libgf.sourceforge.net, on a CVS distribution. Prior to downloading a copy of libGF from CVS, the following steps must be performed to ensure that the user has a working CVS client from which to pull the libGF source code. Users requiring the software that is specified for installation may obtain a copy, if available, through the Naval Postgraduate School Help Desk, or they may obtain the appropriate software from the appropriate Internet URL.

1) Install SSH Secure Shell (e.g., self-executable SSGWinClient-3.1.0-build235.exe)
2) Install Python (e.g., self-executable Python-2.2.1.exe)
3) Install WinCVS (v 1.3 works)
4) Run WinCVS (Programs>GNU>WinCVS)
   a) In Admin>Preferences
      i) On the General tab,
         (1) Change Authentication to → ssh
         (2) Click on Settings (next to Authentication)
            (a) Check the 'if ssh is not in the PATH' box
            (b) Browse to/Enter the executable for ssh (e.g., ssh2.exe)
         (3) Change Path to →/cvsroot/libgf/
         (4) Change Host Address to →cvs.libgf.sourceforge.net
         (5) Change Username to →*yourUsername(developer)* or *anonymous(non-developer)*
         (6) Ensure CVSROOT is →
            yourUsername@cvs.libgf.sourceforge.net:/cvsroot/libgf/ or
            anonymous@cvs.libgf.sourceforge.net:/cvsroot/libgf/
      ii) Check the 'Show CVS console' checkbox
   b) On the WinCvs tab, provide a directory in HOME to store the CVS settings file

2. Downloading and Installing Software Prior to Compilation

In order to use libGF, the user must first extract a copy of the source, in order to compile the source into the needed libraries. Prior to compilation, all libGF files must be extracted and correctly installed (as explained below), and the Microsoft® DirectX® Software Development Kit must also be installed. Do all of this prior to attempting to compile the libGF libraries.

1) From WinCVS, Go to Create>Checkout module…
   a) Type libgf into 'Module name and path on the server'
   b) The CVS console window will pop up. It should say:
      'Host key not found from database. […] Are you sure you want to continue connecting (yes/no)?'
      i) Type yes and hit enter
      ii) You should see the files being downloaded in the bottom pane of the WinCVS window
      iii) When the download is done (you will see *****CVS exited normally with code 0 ***** in the bottom pane following the downloaded files), close the dos CVS console window
2) Install dependency libraries and example data
   a) Go to the URL: http://sourceforge.net/projects/libgf .
   b) Download the current version of libgf-dependencies. Unzip this download into the libgf directory (c:\libgf), ensuring that full path information is used.
   c) Download the current version of libgf-example-data. Unzip this download into the libgf\example directory (c:\libgf\examples), ensuring that full path information is used.
3) Install the Microsoft® DirectX® 8.1 (or higher) Software Development Kit (SDK).

3. Setting up Visual C++® 6.0 for libGF Development

In addition to extracting and installing all needed files and dependencies, Visual C++® 6.0 should be correctly installed and set up for compilation, since the libGF file structure includes the Visual C++® 6.0 workspace and project files. Because Microsoft® DirectX® is required for compilation, the user must compile in a Microsoft® environment, so Visual C++® 6.0 seemed the correct choice. Setting up Visual C++® 6.0 correctly, as explained below, ensures that all needed libraries are referenced.

1) Install Visual C++® 6.0 (Standard install)
2) Open Visual C++® 6.0
   a) In Tools>Options>Directories tab
      i) In 'Show directories for:' click on Include files
         (1) Add the Include directory for the Microsoft® DirectX® SDK (e.g., c:\DXSDK\include)
         (2) Add the Include directory for gfLib (c:\libgf\inc)
         (3) Add the Include directory for gfLib external dependencies (c:\libgf\ext\inc)
      ii) In 'Show directories for:' click on Library files
         (1) Add the Library directory for the Microsoft® DirectX® SDK (e.g., C:\DXSDK\LIB)
         (2) Add the Library directory for gfLib (c:\libgf\lib)

(3) Add the Library directory for gfLib external dependencies (c:\libgf\ext\lib)

***VERY IMPORTANT–the DirectX® Include directory and the DirectX® Library directory need to be the first folders in their respective Options tab

4. Setting up the Environment Variables

The libGF libraries dynamically link required .dll libraries at runtime as needed; in order to ensure this happens, the location of all dynamic link libraries needed at runtime must be in the path.

1) In Start>Control Panel>System>Advanced tab>Environment Variables:
   Add to the path: C:\libgf\ext\bin

5. Building the libGF .lib Files

Finally, once all of the above steps (1 through 4) are accomplished, the libGF libraries can be built as one batch process. Upon completion, the libGF libraries will be available in the libgf\lib directory and will be available for use.

1) Open the gf.dsw workspace in c:\libgf\src
2) Go to Build>Batch Build…
3) Ensure all check boxes are selected
4) Click Build

6. Building the Example Programs

In order to see example applications for libGF and all of its APIs, an examples workspace is available which provides the ability to batch build all of the sample applications. Upon completion, the libGF example applications will be available through either the examples workspace or their individual workspaces in the libgf\src\examples*exampleName* directory.

5) Open the examples.dsw workspace in c:\libgf\src\examples
6) Go to Build>Batch Build…
7) Ensure all check boxes are selected
8) Click Build

B. A BASIC LIBGF APPLICATION

A basic libGF application, at a minimum, consists of a gfSystem as shown in Figure 64. The system is instantiated and initialized before all other classes. All other class members created in the main function are then instantiated and initialized. The gfSystem is then configured and run. gfSystem uses callback methods, so no member instantiation code should follow sys->Config(). The basic application displayed in Figure 70 does not, by itself, do anything noticeable.

```
int main( int argc, char **argv)
{
  gfSystem *sys = new gfSystem("mySystem");
  sys->Init(argc, argv);

 //add all additional code here

  sys->Config();

  sys->Run();

  sys->Exit(0);

  return 0;
}
```

Figure 70. A basic libGF application consists of a gfSystem

C. ADDING MEDIAPATHS

In order to prevent the user having to type path names in for all files needed, and to allow the user the flexibility of changing the path used once as opposed to changing file names many times within a program, libGF allows users to set a list of gfMediaPaths from which to look for all files. The program first looks in the current directory, as expected, but if it does not find the file in the current directory, the program then looks through the list of media paths, in the order provided, until it finds the first copy of the file asked for. gfMediaPath implementation is depicted in Figure 71.

```
  gfMediaPath("C:\\libgf\\src\\examples\\data\\Models\\");
  gfMediaPath("C:\\libgf\\src\\examples\\data\\Textures\\");
```

Figure 71. Implementation of gfMediaPath

If, after setting the gfMediaPaths, the programmer uses a filename that is within one of the specified path entries, the method to manually extract the full filename (path plus filename) is depicted in Figure 72.

```
  char pathPlusFileName[256];
gfGetFullFileName("someFile.xml", pathPlusFileName);
```

Figure 72. Manual use of gfMediaPath

D. ADDING A WINDOW

In order to view the 3D world, the programmer has to create a gfWindow. The analogy to the gfWindow is that of having a box over one’s head and needing a hole in the box to be able to see. Keep in mind at this point, however, that the hole in the box, by itself does not present a picture. The hole is not oriented, and even if it were, nothing is there to look at. Implementation of a gfWindow is shown in Figure 73.

```
//Make a window
gfWindow *mainWindow = new gfWindow(sys,"mainWindow");//, true);
        //note—the use of the optional Boolean on the end is for full screen mode,
        //  which works if the window size is a standard size
        //  (640x480{default}, 800x600, 1024x768, 1152x864, 1280x1024, 1600x1200)

//The following is not required, as it is defaulted (to 640x480)
mainWindow1->WinSize(0, 800, 0, 600);   //values are: left, right, top, bottom
```

Figure 73. Creating a gfWindow

E. ADDING AN OBSERVER

In order to see anything through the hole in the box, the application must create a gfObserver. This observer is analogous to the person looking at the world; Figure 74 depicts the creation of the gfObserver. Although there is now a viewer, there is still no field of view, no direction to look at, and nothing in the world. The observer will tie several other class members together as they are created.

```
//Make an observer
gfObserver *obs = new gfObserver("MainObserver");
```

Figure 74. Creating a gfObserver

F. ADDING A CHANNEL

A gfChannel is the field of view given to the observer; the gfChannel provides the viewing frustum. Consider that the hole in the box from the gfWindow analogy does not provide a picture by itself—it has to be pointed in a particular direction; additionally, the distance from the viewpoint (the eyes) to the hole in the box also makes the view different, as well as the size of the hole. Figure 75 describes gfChannel instantiation and settings.

```
  //make a channel for the window
  gfChannel *mainChannel = new gfChannel("mainChannel");

  //set the window in which the channel will be displayed
  mainChannel->SetWindow("mainWindow");

  //… and give the observer a field of view
  obs->Channel("mainChannel");

//The following are not needed settings, as they are all defaulted

  mainChannel->SetFOV(hFOV, vFOV); //set horizontal and vertical field of view in
                                   //  degrees; default is 45 degrees horizontal and vertical = -1
                                   //  (sets vertical based on horiz FOV and screen dimensions)

  mainChannel->NearFar(nearDistance, farDistance) //sets the near and far clipping
                                                  //  planes

  mainChannel->ClearColor(0.5f, 0.5f, 0.9f, 0.f);  //the viewing background color when
                                                   //  nothing is present
```

Figure 75. Creating a gfChannel and setting several of its parameters

G. ADDING A SCENE

A window and an observer, with its channel, provide a portal through which the user can view what is "outside the box", but the user will not see anything if nothing is there. The world that the user is looking at through the gfWindow and via the gfObserver and gfChannel is the gfScene. A gfScene must be added to the application, as in Figure 76. Note that the scene represents the world that the user is looking at, but this world is still empty until objects are added.

```
  //Make a scene
  gfScene *scene = new gfScene("Scene");

  //… and give the observer the scene to look at
  obs->Scene("Scene");
```

Figure 76. Creating a gfScene

H. ADDING AN ENVIRONMENT

While the scene provides the world to put objects in, if the user wants the objects to experience environmental conditions, (such as time-of-day lighting, shadows, or fog) a gfEnvironment must be added to the scene, as shown in Figure 77. Objects in the scene can then be added to the environment, vice adding them to the scene, in order to render those objects using the gfEnvironment's effects. The environment is not only given to the scene, it is also given to the observer so that the observer's view will reflect environmental settings on visible objects.

```
  //Make an environment
  gfEnvironment *env = new gfEnvironment("Environment");

  //add the environment to the scene
  scene->AddEnvironment(env);

  //set the environment to the observer so the observer can view objects in the
  //  scene using correct environmental conditions
  obs->SetEnvironment(env);

//**** making a (sun)light and adding it to the environment

  //Make a light
  gfLight *light1 = new gfLight("light1");
  gfPosition *lightPos = new gfPosition(50.f, 1000.f, 20.f, 0.f, 0.f, 0.f );
  light1->Position(lightPos);
  env->AddLight("light1");
```

Figure 77. Creating a gfEnvironment, adding it to the scene, setting it to the observer

I. ADDING A DATABASE MANAGER

Prior to creating gfObjects to put into the visible 3D world, if those objects are going to be made of geometry that will be loaded through gfDatasets, a gfDBManager must be created. The gfDBManager initializes database management tools and also allows for extensibility by loading available plugins for different file formats. The gfDBManager does not only handle managing 3D file formats; it also handles retrieval of information from texture files.

```
  //Make a database manager to load files
  gfDBManager *dbm = new gfDBManager();
```

Figure 78. Creating a gfDBManager to open and manage file information

J. ADDING AN OBJECT

Adding objects to the visible 3D world that the user can see is a culmination of the window and channel, the scene and environment, and possibly the database manager. The first step in making an object visible is to instantiate a gfObject. By itself, however, a gfObject has no geometry; for a basic gfObject, the geometry must come from a model file created in a 3D modeling application. File formats supported include .3ds, .flt, and .wrl. In order to use the geometry from the 3D file, several steps must occur. First, the user must create a gfDataset into which to load the model data; second, the user must load the file into the gfDataset; and, finally, the dataset must be added to the object. The gfObject then has geometry, but is still not visible until added to the scene, either by adding it directly to the gfScene, or by adding it to the gfEnvironment, which is added to the scene. All of these steps are depicted in Figure 79.

```
  //Instantiate an object
  gfObject *visibleObject = new gfObject("visibleObject");

  //Make a dataset for the visible object's geometry
  gfDataset *objectDS = new gfDataset("objectDS");

  //Load the object's geometry from a 3D file into the dataset
  objectDS ->LoadFile("visibleObject.flt");

  //Add the dataset to the object
  groundObject->AddDataset("groundDS");

//*** Here, there are one of two options; either…

  //add the object to the environment
  env->AddObject("groundObject");
//*** or…

  //add the object directly to the scene
  scene-> AddObject("groundObject");
```

Figure 79. Creating a gfObject, loading its geometry, an adding it to the scene or environment

K. ADDING A MOTION MODEL

To enable the player to move around the environment, it must have a motion model to provide adjustments to its position in the virtual environment. This thesis deals mainly with using character animation, so it is assumed that the motion model to use in the application is gfMotionHuman. In order to add a new motion model, follow the steps listed in Figure 80.

```
gfMotionHuman *motionPtr = new gfMotionHuman(motionNameStr, bFlipJoystick);
motionPtr->SetInput(motionInputStr);

gzRefPointer<gfPosition> pos = new gfPosition(positionX, positionY, positionZ,
                                              positionH, positionP, positionR);

motionPtr->Position(pos);

motionPtr->SetWalkingSpeed(walkingSpeed);
motionPtr->SetRunningSpeed(runningSpeed);
motionPtr->SetWalkRunThreshold(walkRunThreshold);
motionPtr->SetRotationInterval(rotationInterval);
motionPtr->SetGlanceInterval(glanceInterval);
motionPtr->SetSideStepInterval(sidestepInterval);
motionPtr->SetForwardVelocity(walkingSpeed);
motionPtr->SetRotationVelocity(rotationVelocity);
motionPtr->SetStepUpHeight(stepUpHeight);
```

Figure 80. Creating a new gfMotionHuman motion model

L. ADDING A PLAYER

Adding controlled movement to objects in the 3D world and, in particular, to the observer looking at the scene (via a gfObserver) is accomplished through the use of gfPlayers. Again consider that the observer looking through the hole in the box. The observer has a window (the gfWindow), he has a viewing frustum (the gfChannel), and he has a view of the world outside the box, which he can see (the gfScene and gfEnvironment, and all gfObjects). What he does not have is the ability to orient his view. The gfPlayer, with the help of the gfMotion model, allows the observer to tether himself to a player in order to be moved around the scene via input. The gfPlayer class is not only useful for giving the observer a means to move around the scene, it also gives any other dynamically moving objects a means to move around as well. The gfPlayer ties movement to the visible objects in the scene by adding visible objects, such as gfObjects, to itself. Figure 81 depicts creating the gfPlayer, adding the motion model that the player will use, adding visible scene objects (gfObjects) to the player, and tethering the observer to the player.

```
//Make a player
gfPlayer *player = new gfPlayer("Player");

//set the motion model which will control the player’s movement
```

```
player->SetMotion(motion);

//add all visible objects; the object position will be the same as the gfPlayer position…
player->AddVisObj(visibleObject);

//…so if the object needs to be offset, do a SetOffset() on the object
 gfPosition * visObjOffset = new gfPosition(0.0f, 1.0f, 0.0f, 90.0f, 0.0f, 0.0f);
 visibleObject->SetOffset(visObjOffset);

//If this is the player that the observer will be tied to in order to move around, create
//  an offset position and tether the observer to the player
gfPosition *obsPos = new gfPosition(0.f, 1.2f, 1.2f, 0.f, 0.f, 0.f);
obs->TetherOffset(obsPos);
obs ->TetherMethod(gfObserver::GFOBS_TETHER_FIX_PCS);
obs ->TetherPlayer(player);
```

Figure 81. Creating a gfPlayer, adding a motion model, adding a visible object, and tethering the observer

M. ADDING A GFGRAPHICS

There are several functions that are nice for the programmer to have at hand in order to change the graphic representation and screen information. Displaying in wireframe can show the programmer what is being drawn behind objects, and can render much faster. The ability to turn texturing off and without major code revision is also very useful for optimization during development. These states can be switched on and off through the use of a gfGraphics. In addition to these optimizations, a gfGraphics member will also automatically ensure that backface culling is in effect, so that the scene is rendered efficiently. Other possible future additions to gfGraphics include: the ability to turn on/off an informational HUD that shows pertinent development information, such as frames per second; the ability to zoom in and out (through use of glFrustum commands or access to the frustum); and stencil buffer effects such as binoculars or view-through-a-window (i.e., the dash and roof of a car) outlining frames. Figure 82 shows how to implement the gfGraphics class.

```
//Make a gfGraphics (allows wireframe and other functionality)
gfGraphics *graphics = new gfGraphics("graphics");

//to set the scene to wireframe (use false to turn back off)
graphics->SetWireframeMode(true);

//to turn off all textures applied in the scene (use true to turn back on)
graphics->SetTextureMode(false);
```

Figure 82. Creating a gfGraphics

N. ADDING A GFGUI

Once the user has everything he wants built into the application and roughly where he wants, he may find, when he runs the application, that visible objects are in slightly incorrect locations, or that he wants to adjust the effects of the environment, or that he wants to move the observer's offset from the player. In all of these cases, and many other situations, the user can use the gfGUI class to make run-time changes, in order to find the best settings to use in an application. Use of the gfGUI is depicted in Figure 83.

```
gfGUI* gui = new gfGUI();
```

Figure 83. Creating a gfGUI

O. DISPLAYING CONSOLE NOTIFICATIONS

libGF has a notification system which sends notifications such as warnings, notices, and debug information to the console; this allows the libGF engine, as well as the programmer, to pass important information to the user. However, information needed due to fatal error can be significantly different than general system notifications, and the programmer may not always want the end user to see all information, while he may, himself, want to see all messagees. libGF allows the programmer to set the level of console notifications that he wants to receive, so that in one instance he receives all messages, such as warnings, while in another instance, he receives no notifications (other than fatal messages). This allows the programmer to see all warning or notification messages while debugging and testing a program, but also allows him to turn all of those messages off for a final product. The command to set the notification level desired and the method to add additional notifications are displayed in Figure 84; the notification levels allowed and their corresponding enumeration value are depicted in Figure 85.

```
gfSetNotifyLevel(GF_DEBUG);

gfNotify(GF_WARN, "Warning message number %d, same format as printf", 1);
```

Figure 84. Setting the console notification level and sending messages to the notification system

```
enum gfNotifySeverity {
   GF_ALWAYS=0,
   GF_FATAL=1,
   GF_WARN=2,
```

```
    GF_NOTICE=3,
    GF_INFO=4,
    GF_DEBUG=5
};
```

Figure 85. The gfNotification levels

P. SUBSCRIBING TO THE GFSYSTEM

In many instances, members of libGF classes with no ties to other members need to pass necessary information. An important example of this necessity is that the gfSystem needs to tell all libGF members when a cycle has completed, so that they can perform necessary end-of-cycle steps. When creating new classes, either as an application programmer or when developing new libGF functionality, the programmer needs to be aware of how to subscribe classes to the gfSystem, and how to make similar subscriptions between class members.

In order to facilitate message passing between libGF class members, gfBase—the base class and runtime type identifier for almost all libGF classes—allows members to subscribe to other members, so that the subscriber can listen to all notifications by the sender. In order to listen to the gfSystem, libGF members must subscribe to the gfSystem as follows:

- **In the header (.h) file:**

The class which will listen to the gfSystem must inherit from gfBase or from a class that inherits from gfBase. In addition, the class must redefine the virtual *onNotify* method (from gfBase). Figure 86 depicts both of these requirements.

```
class yourClassName : public gfBase {

virtual void onNotify(gzNotifyMessage *message);

};
```

Figure 86. Inheriting from gfBase and redefining onNotify() in the .h file

- **In the source (.cpp) file:**

In the constructor, the class must add the system as a notifier to the class, as such:

```
//make the system a notifier to this class
gfSystem *sys = (gfSystem*)SystemList->Get(0);
```

```
        AddNotifier(sys);
```
Figure 87. Adding the system as a notifier to a class

In onNotify, the following code, in Figure 88, allows the listener (the subscribing class) to listen for particular system messages:

```
void gfWindow::onNotify( gzNotifyMessage *message )
//a notification has been captured from one of this member's notifiers; this
// (redefinition of the virtual void) function will handle the notification
{
  gzTypeInterface *sender = message->getSender(); //gets the message sender
  gzString command = message->getCommand();     //gets the command sent

  if (sender->isExactType(gfSystem::getClassType())) //if the sender is the
                                                     //system; this member might subscribe
                                                     // to more than one notifier
  {
    if (command == "xxxxxxx")          //if the command is what this
                                       //class is looking for
    {
      your code here;                  //Then perform this code
    }
  }
}
```
Figure 88. Class specific definition of onNotify() in the .cpp file

The commands available from gfSystem include: frame, tick, configure, and exit. If listening for notification by other classes, the subscribing member must know what commands it may receive in order to process them. Getting the data from the message (passed into onNotify()) in order to use it, when needed, is done as such:

```
ClassToCastDataTo *data = (ClassToCastDataTo *)message->getData();
```
Figure 89. Casting notification data passed into onNotify()

Further example of the subscription of class members to notifying class members can be seen in the Networks section of Chapter II.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B: EXPERIMENT QUESTIONNAIRE

***Please read first*:** The following experiment and questionnaire are conducted completely anonymously. Nothing you do or answer will be related back to you in any manner. Thank you for your assistance. Please begin below the solid line and hand to the proctor when you reach "Stop Here". You may ask questions at any time.

Subject Number ________ (proctor use only)

Preliminary questions:

1. Do you have any history of epilepsy? Yes / No

2. Are you prone to simulator sickness? Yes / No

3. Do you require corrective lenses? Yes / No

4. What is your vision uncorrected?

5. Do you have any other history of eye disease or injury?

6. How often do you use a computer on a daily basis? (Check one.)

❍ 0-2 hours ❍ 2-4 hours ❍ 4-6 hours ❍ 6-8 hours ❍ greater than 8 hours

7. Have you ever used virtual environment for training or entertainment? Yes / No

8. If yes, did you use a head-mounted display (HMD)? Yes / No

9. What First Person Shooter (FPS) games are you familiar with?

____

____

____

____

10. How many hours, on average, do you play FPS games? (Check one)

❍ 0-2 hours ❍ 2-4 hours ❍ 4-6 hours ❍ 6-8 hours ❍ greater than 8 hours

❍ Day ❍ Week ❍ Month (Check one)

11. How would you rate your level of training in Mobile Operations on Urban Terrain (MOUT)? (Check one.)

❍ novice ❍ average ❍ advanced ❍ instructor ❍ expert

12. List all exercises and locations that you have conducted or been involved with MOUT or CQB.

____
____
____
____

13. About how many hours of MOUT training have you received? _______________

14. Evaluation task:

You will be provided with several building clearing tasks, each with several segments. Before each segment, you will be briefed on what you are expected to do and which path you need to take. The amount of time required from start of each segment until the end of that segment will be recorded and used to help determine the ease of use for each interface device for MOUT virtual environment training. You will be asked to perform each set of tasks twice per interface device, and you will perform the experiment using three different interface devices (Keyboard/Mouse, Gamepad, Head Mounted Display with Instrumented Rifle), chosen in a random order.

The Goal:
The goal of this experiment is not to measure your overall building clearing skills, but instead to try and determine which interface allows you to maneuver in the virtual environment quicker, more reliably, and more realistically. WE ARE EVALUATING THE SYSTEM, NOT YOUR PERFORMANCE.

Your Resources:
-Overhead map of town, with annotated paths

The Tasks:

You will conduct a series of six tasks. Each task will consist of three segments. You will be briefed on the path to use from the start of each segment through engagement of enemy targets. Time keeping will start on the commencement of each segment and will terminate when the final enemy target is successfully shot (for that segment). Questions will be asked at the end of each segment, followed by the proctor ensuring that you are in place for the next segment.

The tasks include:

Clear a building using keyboard and mouse
Clear a building using a gamepad setup

Clear a building using a Head-Mounted Display (HMD) and Instrumented Rifle (joystick enabled) setup

The tasks will be conducted in a random order, but each of the tasks (keyboard and mouse, gamepad, and HMD/rifle) will be conducted two times sequentially.

In all of the three tasks, while the path will be predetermined and the proctor will be directing you (prior to each segment) on where to go, you will be expected to conduct good building/room clearing techniques. Ensure you are aware of your field of fire and that you move tactically, taking into account all danger areas such as doors, windows, open areas, linear danger areas, and constricted areas/choke points.

Do you have any questions?

Stop here.
Notify the proctor.

(proctor use only)

Check for history of epilepsy or proneness to simulator sickness.

Perform familiarization walk-through of town. Proctor will perform walk-through, but subject is allowed to have the walk-through stopped to allow longer view in any area.

Segments are to be performed in a random order and each conducted twice.
Order selected for input devices:
______ Keyboard & Mouse
______ Gamepad
______ HMD and Instrumented Rifle

Keyboard and Mouse Task:

First run:
First Segment:
Show the participant Building 1, Segment 1

Time to complete segment: __________

Second Segment:
Show the participant Building 1, Segment 2

Time to complete segment: __________

Would you rate this segment easier or harder than the last segment? Easier/Harder than 1

Third Segment:
Show the participant Building 1, Segment 3

Time to complete segment: __________

Would you rate this segment easier or harder than each of the other segments? Easier/Harder than 1
Easier/Harder than 2

Second run:
First Segment:
Show the participant Building 1, Segment 1

Time to complete segment: __________

Second Segment:

Show the participant Building 1, Segment 2

Time to complete segment: ___________

Would you rate this segment easier or harder than the last segment? Easier/Harder than 1

Third Segment:

Show the participant Building 1, Segment 3

Time to complete segment: ___________

Would you rate this segment easier or harder than each of the other segments? Easier/Harder than 1
Easier/Harder than 2

Gamepad Task:

First run:

First Segment:

Show the participant Building 2, Segment 1

Time to complete segment: ___________

Second Segment:

Show the participant Building 2, Segment 2

Time to complete segment: ___________

Would you rate this segment easier or harder than the last segment? Easier/Harder than 1

Third Segment:

Show the participant Building 2, Segment 3

Time to complete segment: ___________

Would you rate this segment easier or harder than each of the other segments? Easier/Harder than 1
Easier/Harder than 2

Second run:

First Segment:

Show the participant Building 2, Segment 1

Time to complete segment: ___________

Second Segment:
Show the participant Building 2, Segment 2

Time to complete segment: ___________

Would you rate this segment easier or harder
than the last segment? Easier/Harder than 1

Third Segment:
Show the participant Building 2, Segment 3

Time to complete segment: ___________

Would you rate this segment easier or harder
than each of the other segments? Easier/Harder than 1
Easier/Harder than 2

HMD and Instrumented Rifle Task:

First run:
First Segment:
Show the participant Building 3, Segment 1

Time to complete segment: ___________

Second Segment:
Show the participant Building 3, Segment 2

Time to complete segment: ___________

Would you rate this segment easier or harder
than the last segment? Easier/Harder than 1

Third Segment:
Show the participant Building 3, Segment 3

Time to complete segment: ___________

Would you rate this segment easier or harder
than each of the other segments? Easier/Harder than 1
Easier/Harder than 2

Second run:
First Segment:
Show the participant Building 3, Segment 1

Time to complete segment: ___________

Second Segment:
Show the participant Building 3, Segment 2

Time to complete segment: ___________

Would you rate this segment easier or harder
than the last segment? Easier/Harder than 1

Third Segment:
Show the participant Building 3, Segment 3

Time to complete segment: ___________

Would you rate this segment easier or harder
than each of the other segments? Easier/Harder than 1
Easier/Harder than 2

(proctor use only)

CQBSIM Set up:
Ensure CQBSim and all supporting files and folders (from the same directory) are copied into one directory on the hard drive.
Keyboard and Mouse setup:
Run CQBSim.exe.
From the CQBSim GUI, on the Window tab, ensure the ‘Top Window Position’ and “Left Window Position’ are set to 0, and that the ‘Horizontal Resolution’ and ‘Vertical Resolution’ are set to 1280 and 1024, respectively.
Ensure that the ‘Full Screen’ box is checked.
Switch to the System tab and ensure that the DirectX GUI is checked.
Press ‘Run App’.
From the DirectX GUI, ensure the keyboard and mouse are correctly mapped.
Press ‘OK’.
Joystick setup:
Run CQBSim.exe.
From the CQBSim GUI, on the Window tab, ensure the ‘Top Window Position’ and “Left Window Position’ are set to 0, and that the ‘Horizontal Resolution’ and ‘Vertical Resolution’ are set to 1280 and 1024, respectively.
Ensure that the ‘Full Screen’ box is checked.
Switch to the System tab and ensure that the DirectX GUI is checked.
Switch to the Motion tab, and ensure that ‘Flip Joystick Input’ is deselected.
On the Motion tab, reduce the ‘Rotational Velocity’ to 0.5.
Press ‘Run App’.
From the DirectX GUI, ensure the joystick is correctly mapped.
Press ‘OK’.
HMD and Instrumented Rifle setup with 1 Intersense Tracker:
Intersense tracker setup:
Ensure that the serial Intersense tracker is plugged into the serial port of the computer.
Run isdemo32.exe from the Isense31 directory in the supporting files for CQBSim.
Click ‘Close’
Click ‘Detect’
Ensure that the tracker is located (no failure), and click ‘Accept’
Boresight the tracker from isdemo32.exe, then exit.
Rifle setup:
Ensure the wireless receiver is plugged into a USB port and is active.
Ensure that the LAN cable connects the power supply box to the handgrip of the rifle.
Ensure that fresh batteries have been installed in the power supply box.
Ensure the power supply box is on and active.
Ensure the rifle safety switch is set to ‘Semi’.
HMD setup:
Ensure that the HMD power supply is plugged in and is connected to the HMD control box.
Ensure that the HMD data cable is plugged into the HMD control box.

Ensure that a video cable connects the computer and the left eye (common) input to the HMD control box.
Ensure that the computer monitor is connected to either the computer (via DVI cable) or to the Output connection on the HMD control box via video cable.
Ensure that the power button on the HMD control box is turned on (lights up).
Ensure that the computer video resolution is set to 800 x 600.
Ensure that the Intersense tracker is attached to the HMD via the velcro fastener.
Connect all loose cables to prevent injury or equipment damage.
Run CQBSim.exe.
From the CQBSim GUI, on the Window tab, ensure the 'Top Window Position' and "Left Window Position' are set to 0, and that the 'Horizontal Resolution' and 'Vertical Resolution' are set to 800 and 600, respectively.
Ensure that the 'Full Screen' box is checked.
Switch to the System tab and ensure that the DirectX GUI is checked.
Switch to the Motion tab, and reduce the 'Rotational Velocity' to 0.5.
Press 'Run App'.
From the DirectX GUI, ensure the keyboard and mouse are correctly mapped.
Press 'OK'.
HMD and Instrumented Rifle setup with 2 Intersense Trackers (for future addition to experiment):
Intersense tracker setup:
Ensure that the serial Intersense tracker is plugged into the serial port of the computer.
Run isdemo32.exe from the Isense31 directory in the supporting files for CQBSim.
Click 'Close'
Click 'Detect'
Ensure that the tracker is located (no failure), and click 'Accept'
Go to 'Parameters>Station and Sensor Parameters', and ensure that Station 1 is connected to InertialCube 1, and Station 2 is connected to InertialCube 2
Boresight the trackers, then exit.
Rifle setup:
Ensure the wireless receiver is plugged into a USB port and is active.
Ensure that the LAN cable connects the power supply box to the handgrip of the rifle.
Ensure that fresh batteries have been installed in the power supply box.
Ensure the power supply box is on and active.
Ensure the rifle safety switch is set to 'Semi'.
HMD setup:
Ensure that the HMD power supply is plugged in and is connected to the HMD control box.
Ensure that the HMD data cable is plugged into the HMD control box.
Ensure that a video cable connects the computer and the left eye (common) input to the HMD control box.
Ensure that the computer monitor is connected to either the computer (via DVI cable) or to the Output connection on the HMD control box via video cable.
Ensure that the power button on the HMD control box is turned on (lights up).
Ensure that the computer video resolution is set to 800 x 600.

Ensure that the Intersense tracker is attached to the HMD via the velcro fastener.
Connect all loose cables to prevent injury or equipment damage.
Run CQBSim.exe.
From the CQBSim GUI, on the Window tab, ensure the ‘Top Window Position’ and “Left Window Position’ are set to 0, and that the ‘Horizontal Resolution’ and ‘Vertical Resolution’ are set to 800 and 600, respectively.
Ensure that the ‘Full Screen’ box is checked.
Switch to the System tab and ensure that the DirectX GUI is checked.
Switch to the Motion tab, and reduce the ‘Rotational Velocity’ to 0.5.
Press ‘Run App’.
From the DirectX GUI, ensure the keyboard and mouse are correctly mapped.
Press ‘OK’.

Post-Experiment Questions:
HMD Aftereffects

1. The HMD made me feel queasy / nauseous.
❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

2. The HMD is disorienting because of the need to stand in one position (no body rotation).
❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

Realism

1. The scale of objects in the virtual environment felt correct with respect to a real-world environment. (same in all tasks)

❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

2. The movement rate felt correct with relation to real world movement of personnel through a MOUT scenario.

Keyboard and Mouse setup
❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

Gamepad setup
❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

HMD and Rifle setup
❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

3. My Field of View felt correct with relation to the real world.

Keyboard and Mouse setup
❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

Gamepad setup
❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

HMD and Rifle setup
❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

4. Use the following definition of presence to answer this question: “Presence is the feeling that you are truly in the virtual environment, acting and reacting with the environment as though it were real.”

I felt as though I was present in the virtual environment.

Keyboard and Mouse setup
❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

Gamepad setup
❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

HMD and Rifle setup
❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

Experiment Tasks

1. Movement in the task felt as it would in a real environment.

Keyboard and Mouse setup
❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

Gamepad setup
❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

HMD and Rifle setup
❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

2. Accomplishing the segments given by the proctor was easy.

Keyboard and Mouse setup
❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

Gamepad setup
❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

HMD and Rifle setup
❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

3. Maneuver through doors was easy.

Keyboard and Mouse setup
❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

Gamepad setup
❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

HMD and Rifle setup
❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

4. Maneuver around obstructions was easy.

Keyboard and Mouse setup
❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

Gamepad setup
❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

HMD and Rifle setup
❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

5. Tactical movement (ensuring that the subject was able to avoid danger areas and was able to clear areas around corners and through doors in small zones) through the environment was easy.

Keyboard and Mouse setup
❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

Gamepad setup
❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

HMD and Rifle setup
❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

6. Sighting in on and engaging the target was easy.

Keyboard and Mouse setup
❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

Gamepad setup
❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

HMD and Rifle setup
❍Strongly agree ❍Agree ❍Neither agree nor disagree ❍Disagree ❍Strongly disagree

7. What suggestions for improvements of the maneuverability of the three different setups do you have? Please add any other statements you may have concerning this experiment. (If you have a comment on a specific question please provide the question number.):

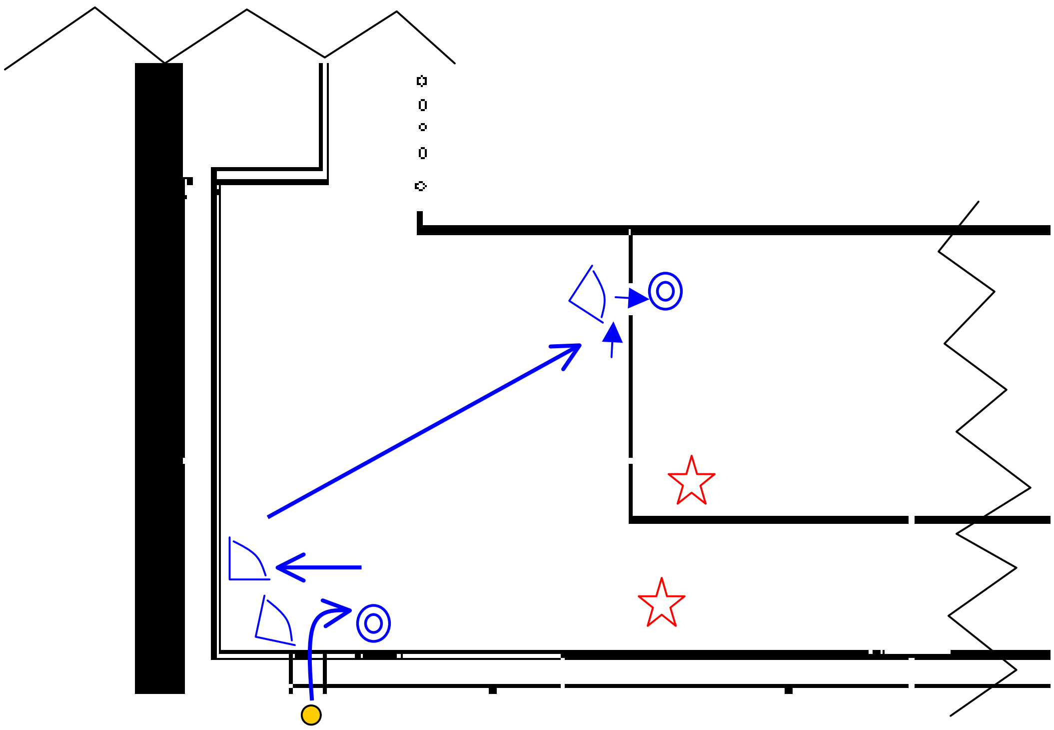
____
____
____
____
____
____
____
____
____
____

Thank you for your participation.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C: EXPERIMENT SCRIPTS

A. BUILDING 1 SEGMENT 1



Key:

= Start

= Engage target here

= Pie off area

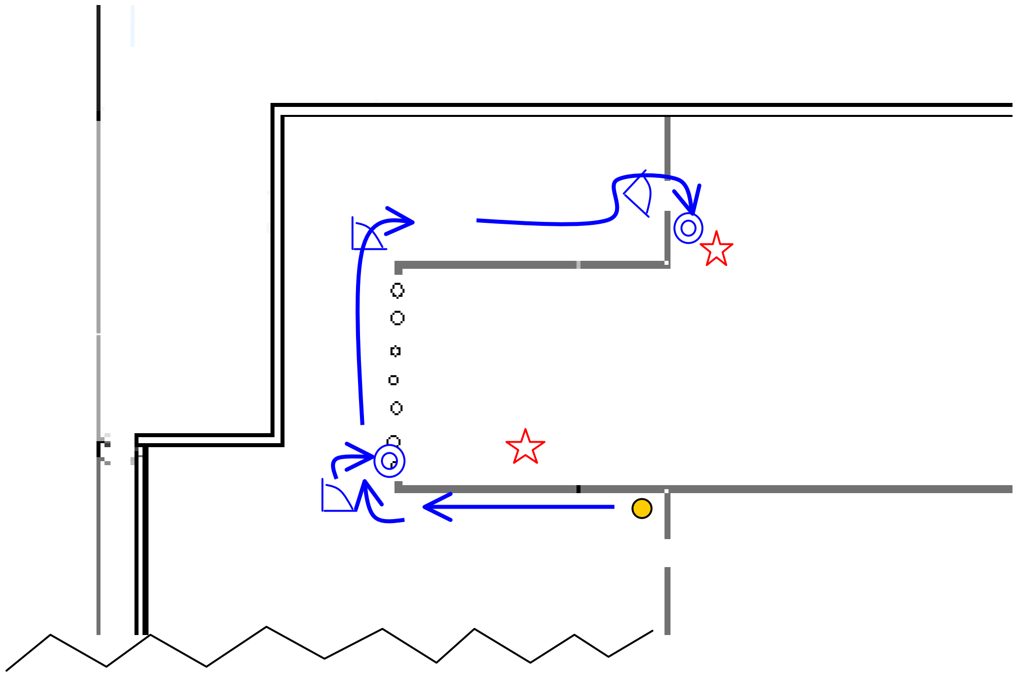
= Target

= Move this direction

Description:

Enter the structure and pie off the area immediately in front of the entrance. Engage the target to your right. Proceed toward the wall on your left, and quickly inspect the area. Cross over to and enter the room on the wall opposite to you. Pie off the entrance to the room and engage the target within the room. Move tactically through the environment as you would in a real environment.

BUILDING 1 SEGMENT 2



Key:

= Start

= Engage target here

= Pie off area

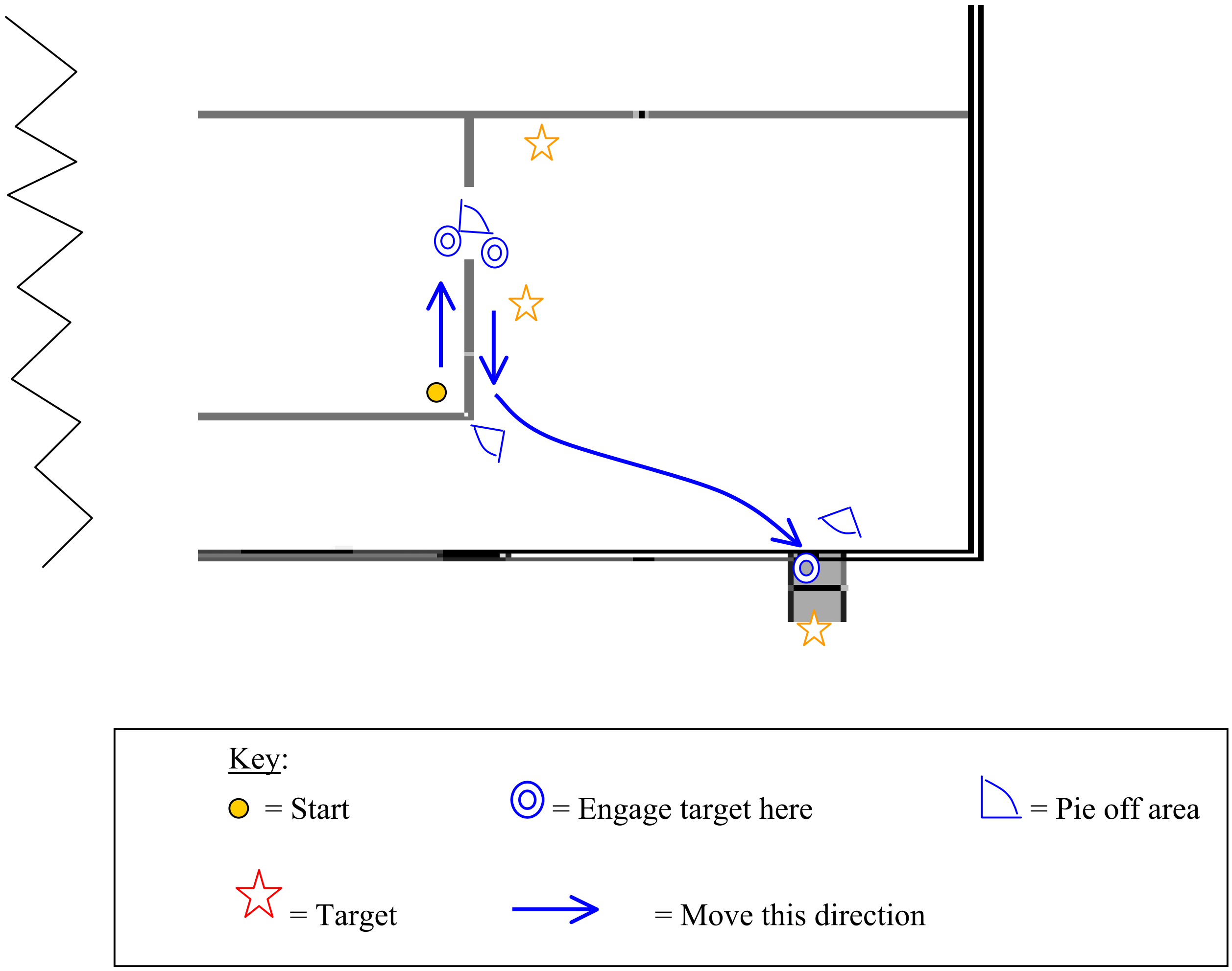
= Target

= Move this direction

Description:

Proceed forward to the corner of the wall. Upon reaching the corner make a right turn and pie off the area. The wall to your right, represented as a series of circles in the map above indicates a barrier of wooden 2x4 beams. Engage the target by firing between the beams. Proceed to the next corner and make a right turn after you pie off the area. Following the contours of the wall to your right, move to the next doorway. Pie off the room and engage the target in the room. Move tactically through the environment as you would in a real environment.

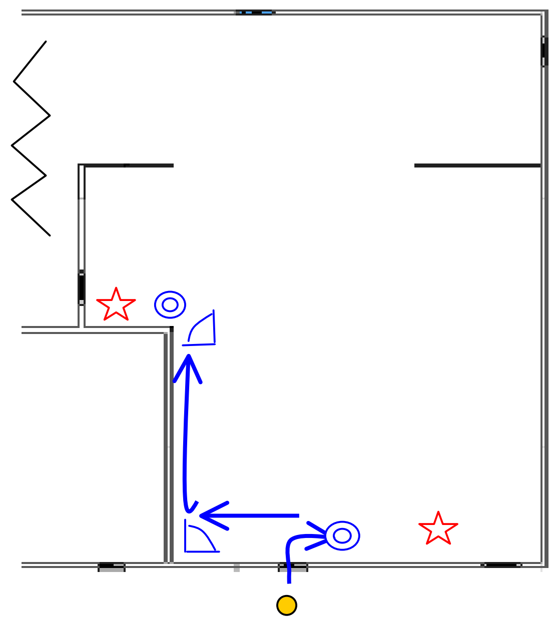
C. BUILDING 1 SEGMENT 3



Description:

Proceed forward towards the entrance to your right. Pie off the area near the entrance and engage the target against the far wall. Continue to pie off the area until you successfully engage the second target in the room. Enter the room and proceed along the wall to your right. Pie off the hallway around the corner. Cross the room toward the wall and proceed to the exit on your left. Pie off the area outside of the exit and engage the target. Move tactically through the environment as you would in a real environment.

D. BUILDING 2 SEGMENT 1



Key:

= Start

= Engage target here

= Pie off area

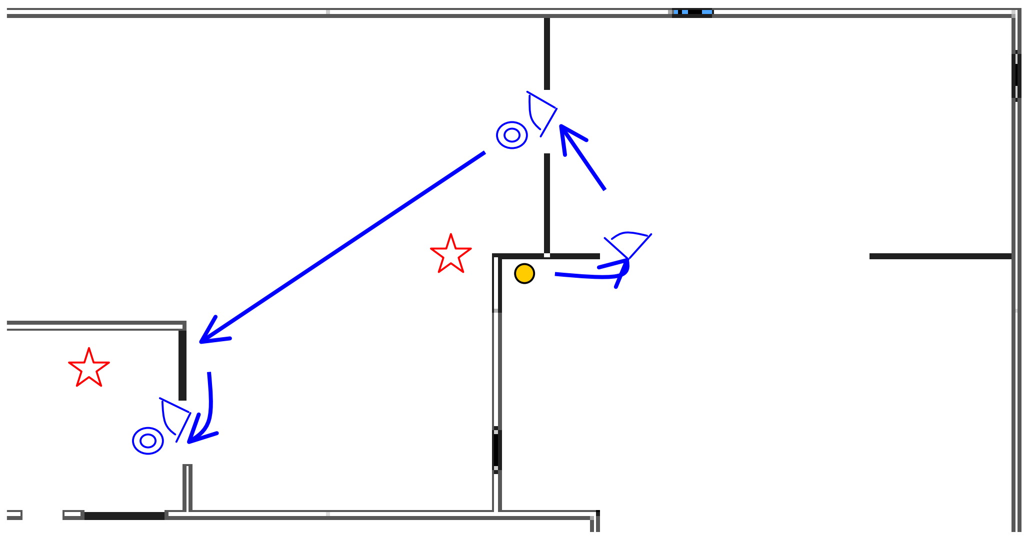
= Target

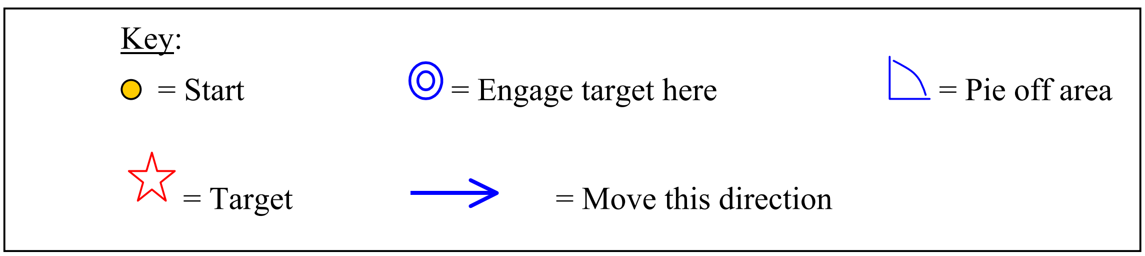
= Move this direction

Description:

Enter the structure and pie off the area immediately beyond the entrance. Engage the target to your right. Proceed toward the wall on your left, and briefly inspect the area. Follow the wall to the corner. Pie off the area around the corner and engage the target to your left. Move tactically through the environment as you would in a real environment.

E. BUILDING 2 SEGMENT 2

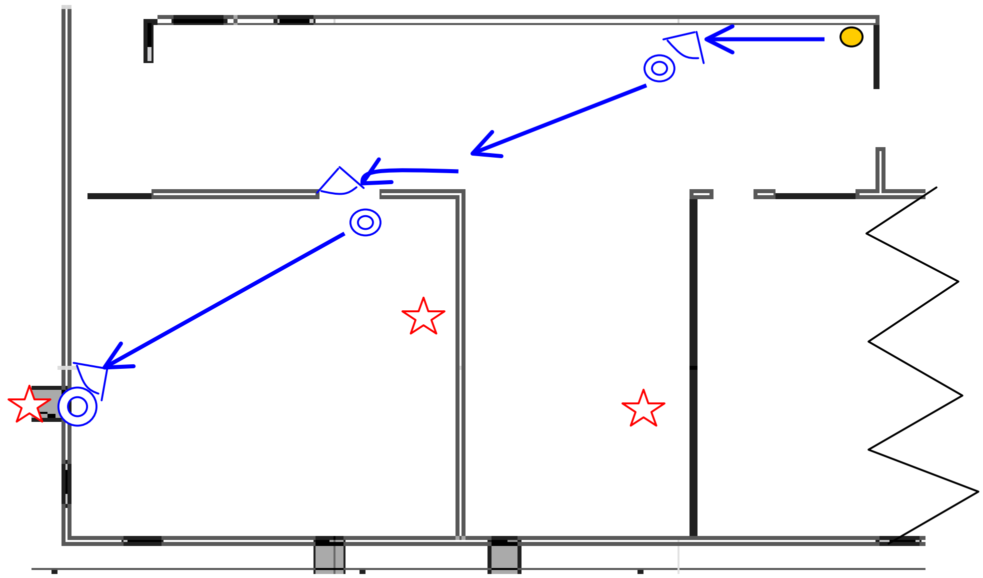




Description:

Move along the wall to your left until coming to the corner. Pie off the area at the corner and proceed along the other side of the same wall until you reach the next corner. Pie off the area around the corner and engage the target. Proceed across the room to the next doorway. Pie off the area in the doorway and engage the target. Move tactically through the environment as you would in a real environment.

F. BUILDING 2 SEGMENT 3



Key:

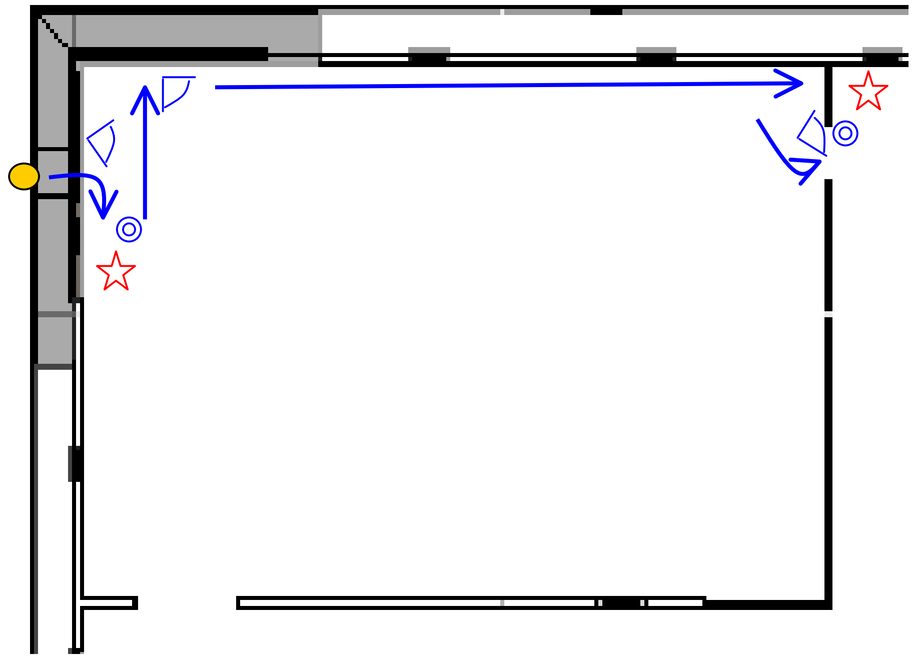
= Start   = Engage target here   = Pie off area

= Target   = Move this direction

Description:

Move along the wall to your right until you see the long corridor off to your left. Pie off the corridor and engage the target. Cross the room and move along to wall to your left until you reach the next doorway. Pie off the room and engage the target. Once clear, move across the room to the exit. Pie off the area outside of the exit and engage the target. Move tactically through the environment as you would in a real environment.

G. BUILDING 3 SEGMENT 1



Key:

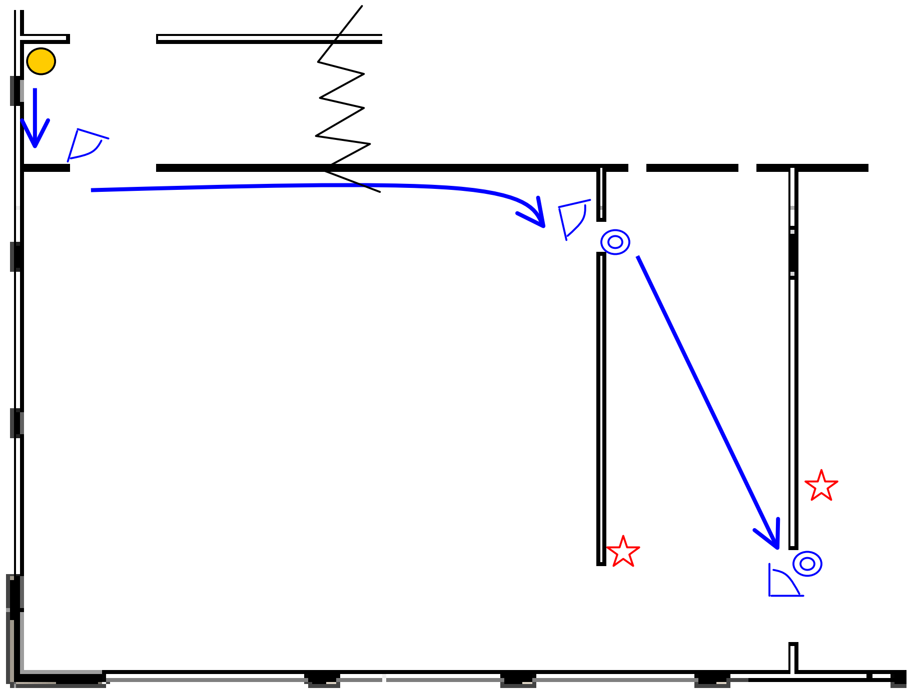
= Start = Engage target here = Pie off area

= Target = Move this direction

Description:

Enter the structure and pie off the area immediately beyond the entrance. Engage the target to your right. Proceed toward the wall on your left, and briefly inspect the area. Follow the wall on your left to the next doorway. Pie off the area through the doorway and engage the target in the corner. Move tactically through the environment as you would in a real environment.

H. BUILDING 3 SEGMENT 2



Key:

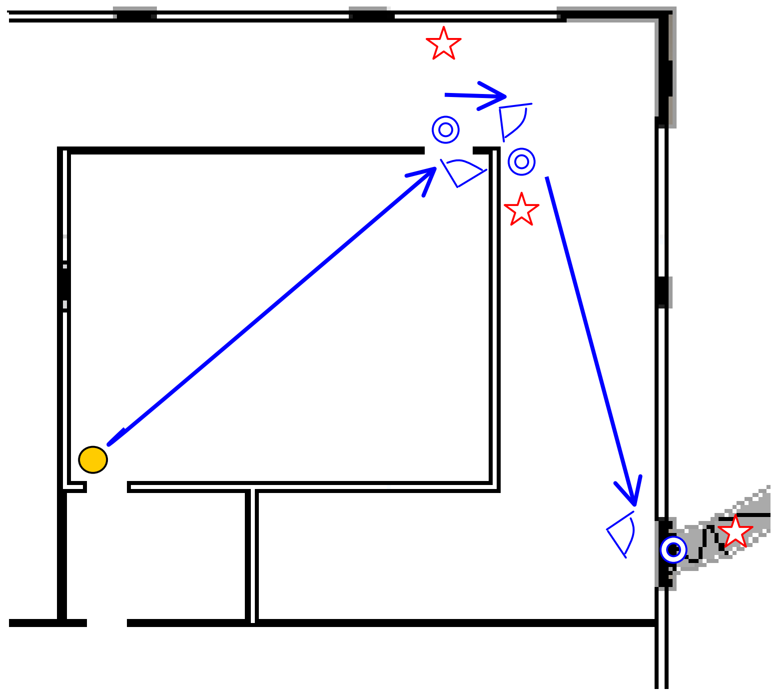
= Start   = Engage target here   = Pie off area

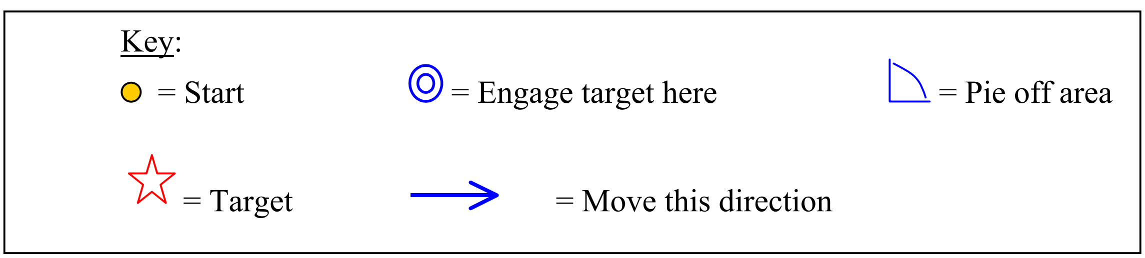
= Target   = Move this direction

Description:

Proceed towards the entrance in front of you. Pie off the area through the entrance. Turn left and proceed along the wall towards the next doorway. Pie off the area through the doorway and engage the target lying along the right wall. Once clear, cross the room and approach the far right doorway. Pie off the area around the doorway and engage the target. Move tactically through the environment as you would in a real environment.

I. BUILDING 3 SEGMENT 3





Description:

Cross the room to the doorway at your far right. Pie off the area through the doorway and engage the target. Turn right and pie off the area around the corner to your right. Engage the target against the right wall. Once clear, proceed down the corridor to the exit on the far left. Pie off the area beyond the exit and engage the target. Move tactically through the environment as you would in a real environment.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Chairman, Code CS
   Computer Science Department, Code CS
   Naval Postgraduate School
   Monterey, California

4. Dr. Rudy Darken
   Modeling of Virtual Environment and Simulations (MOVES) Program
   Naval Postgraduate School
   Monterey, California

5. CDR Joseph Sullivan
   Computer Science Department, Code CS
   Naval Postgraduate School
   Monterey, California

6. Marine Corps Representative
   Naval Postgraduate School
   Monterey, California

7. Director, Training and Education, MCCDC, Code C46
   Quantico, Virginia

8. Director, Marine Corps Research Center, MCCDC, Code C40RC
   Quantico, Virginia

9. Marine Corps Tactical Systems Support Activity (Attn: Operations Officer)
   Camp Pendleton, California